PHP BASICS

Hypertext Preprocessor

PHP (Hypertext Preprocessor) is a popular open-source scripting language used to create dynamic and interactive web pages and applications.

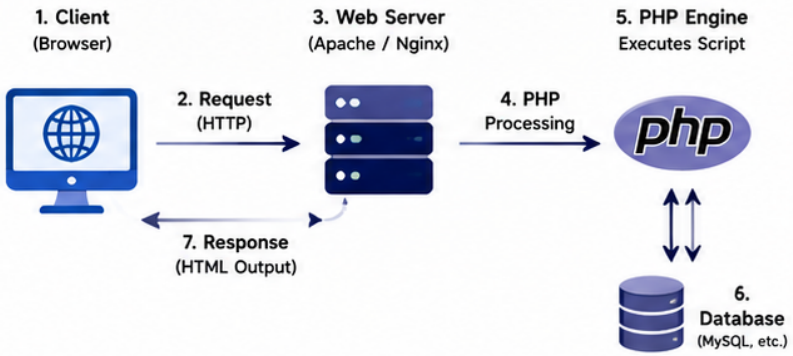
Server-Side
PHP runs on the server, not in the browser.

Web Development
PHP is a powerful language for building websites.

Dynamic Websites
PHP generates dynamic content based on user input, database, or other conditions.

Backend
PHP is primarily used for server-side scripting and backend development.

HOW PHP WORKS



PHP scripts are executed on the server, and the output (usually HTML) is sent back to the browser.

BASIC SYNTAX

```php
<?php
 // This is a PHP comment
 echo "Hello, World!";
 $name = "PHP Developer";
 echo "Welcome, " . $name;
?>
```

- PHP code is written inside `<?php ?>` tags.
- Each statement ends with a semicolon (;).
- Comments start with `//` or `/* */`.

INTEGRATION WITH HTML

PHP can be easily embedded in HTML.

```php
<!DOCTYPE html>
<html>
 <body>
    <h1><?php echo "Hello PHP!"; ?></h1>
    <p>This is a PHP Example.</p>
 </body>
</html>
```

PHP code runs on the server and the result is shown in the browser as HTML.

COMMON USES

- User Authentication & Sessions
- Form Handling & Validation
- Database Operations
- Content Management Systems (CMS)
- REST APIs & Web Services

KEY ADVANTAGES

- Open Source & Free
- Easy to Learn & Use
- Cross-Platform
- Large Community Support
- Works with Many Databases
- Scalable & Flexible

WORKS WITH

HTML · CSS · JS · MySQL

PHP is a server-side scripting language designed for web development. It helps you build dynamic, interactive and database-driven websites.

PHP Installation & Setup

Install PHP and Set Up Your Local Development Environment

```
<?php
echo "Hello PHP!";
?>
```

PHP — Server-Side Scripting Language

Apache — Web Server to Run PHP

XAMPP — Easy PHP + Apache + MySQL Setup

WAMP — Windows Web Environment

Laragon — Lightweight & Fast PHP Environment

VS Code — Best Code Editor for PHP

1. INSTALL PHP USING LOCAL SERVERS

XAMPP
- Download XAMPP
- Install and Start Apache
- PHP is included
- Access: http://localhost

WAMP
- Download WAMP
- Install and Start All Services
- PHP is included
- Access: http://localhost

Laragon
- Download Laragon
- Start Apache
- PHP is included
- Access: http://localhost

Apache (Manual Setup)
- Install Apache
- Install PHP
- Configure httpd.conf
- Place PHP in htdocs folder

2. VERIFY PHP INSTALLATION

Check PHP Version (Command Line)

php -v

```
C:\> php -v
PHP 8.2.7 (cli) (built: Jul  4 2024 10:15:00) (NTS Visual C++ x64)
Copyright (c) The PHP Group
Zend Engine v4.2.7, Copyright (c) Zend Technologies
    with Zend OPcache v8.2.7, Copyright (c), by Zend Technologies
```

Check in Browser

http://localhost/info.php

PHP Version 8.2.7

System	Windows NT 10.0; Win64; x64
Build Date	Jul 4 2024 10:15:00
Server API	Apache 2.0 Handler
PHP Extension	mysqli, pdo, curl, json, mbstring, openssl, etc.
Configuration File	php.ini

Great! PHP is installed and working on your localhost.

3. CREATE YOUR FIRST PHP FILE

index.php

```
<!DOCTYPE html>
<html>
<body>
    <?php
      echo "Hello, PHP! 🎉";
     ?>
</body>
</html>
```

Output in Browser

Hello, PHP! 🎉

http://localhost/index.php

4. RECOMMENDED TOOL

VS Code
- Lightweight
- Extensions Support
- PHP IntelliSense
- Debugging Support

index.php

```
<?php
echo "Hello from VS Code!";
?>
```

QUICK COMMANDS

php -v	→ Check PHP Version
php -m	→ List PHP Modules
php -i	→ PHP Info
php --ini	→ Show PHP Config

COMMON LOCALHOST URLS

http://localhost	→ Default Page
http://localhost/phpmyadmin	→ MySQL Database
http://localhost/info.php	→ PHP Info Page
http://localhost/yourfile.php	→ Your PHP File

FOLDER STRUCTURE (XAMPP)

```
C:\xampp\htdocs\
├── index.php
├── info.php
└── your_project\
    └── file.php
```

TIPS

- ✔ Always keep Apache and MySQL running.
- ✔ Place your projects in htdocs folder.
- ✔ Use meaningful file names.
- ✔ Restart Apache if you change php.ini.

localhost

PHP SYNTAX

The basic rules of writing PHP code

```
<?php
 echo "Hello, PHP!";
?>
```

1. PHP TAGS

`<?php` `?>`

Every PHP script starts with <?php and ends with ?>

```
<?php
 echo "Hello, PHP!";
?>
```

Anything outside PHP tags is treated as HTML.

2. VARIABLES

Variables store data. All variables start with a $ sign.

```
<?php
$name = "John";
$age = 25;
$price = 19.99;
?>
```

Variable names are case-sensitive. $Name and $name are different.

3. STATEMENTS

PHP code is made up of statements. Each statement ends with a semicolon.

```
<?php
 $x = 10;
 $y = 20;
 $sum = $x + $y;
?>
```

Always end each statement with a semicolon (;).

4. COMMENTS

Comments are ignored by PHP engine. Used to explain code.

```
<?php
 // This is a single-line comment
 # This is also a single-line comment
 /* This is
     a multi-line
     comment */
?>
```

Comments help make code more readable.

5. ECHO

echo is used to output one or more strings.

```
<?php
 echo "Hello World!";
 echo "PHP is Awesome!";
?>
```

echo can output multiple values (comma separated).

6. PRINT

print is used to output a string. It returns 1.

```
<?php
 print "Hello World!";
 print("PHP is Awesome!");
?>
```

print can take only one argument (unlike echo).

7. SEMICOLON (;)

Semicolon is used to end every statement.

```
$name = "John"; ✔ Correct
$name = "John" ✘ Incorrect
```

If you forget the semicolon, you may get an error.

8. COMPLETE EXAMPLE

Putting it all together!

```
<?php
 // This is a comment
 $name = "Alice";
 $age = 20;
 echo "Name: " . $name;
 echo "<br>Age: " . $age;
?>
```

PHP SYNTAX FLOW

PHP Tags → Variables → Statements → Semicolon → Output (echo/print) → Comments

QUICK TIP

Write clean, simple and readable code.
Always test your code!

PHP VARIABLES

Store Data, Use Anywhere!

```php
<?php
$name = "John";
echo $name; // John
?>
```

WHAT IS A VARIABLE?

A variable is a container used to store data values.

$variable

COMMON VARIABLES (EXAMPLES)

$name	= "John Doe"
$age	= 25
$email	= "john@example.com"
$price	= 19.99

VARIABLE ASSIGNMENT

Use the = operator to assign a value to a variable.

```php
<?php
$name  = "John";          // String
$age   = 25;              // Integer
$email = "john@example.com"; // String
$price = 19.99;           // Float/Decimal
?>
```

Variable names always start with $ sign, followed by a letter or underscore.

VARIABLE TYPES (DYNAMIC TYPING)

PHP is dynamically typed, which means you don't need to declare the type. PHP detects it automatically.

Type	Example	Value	Description
String	$name = "John"	"John"	Text / Characters
Integer	$age = 25	25	Whole Numbers
Float	$price = 19.99	19.99	Decimal Numbers
Boolean	$isAdmin = true	true	true or false
Array	$colors = ["Red", "Green"]	[...]	List of values
NULL	$user = null	null	No value

DYNAMIC TYPING EXAMPLE

A variable can hold different types of data.

```php
<?php
$data = "Hello";          // String
$data = 100;              // Integer
$daàl = 12.5;             // Float
$data = true;             // Boolean
$data = ["PHP", "MySQL"]; // Array
$data = null;             // Null
?>
```

OUTPUT EXAMPLE

You can use variables in echo or print.

```php
<?php
$name = "John";
$age  = 25;
echo "Name: " . $name . "<br>";
echo "Age: " . $age;
?>
```

Output:

Name: John
Age: 25

VARIABLE NAMING RULES

VALID

```
$name
$_name
$name123
$user_name
```

INVALID

$123name	(starts with number)
$user-name	(no hyphen)
$user name	(no space)
$	(only $ is not allowed)

TIP

Use meaningful variable names to make your code clean, readable and easy to maintain.

KEY POINTS

- Variables store data that can be used multiple times.
- Use = to assign values to variables.
- PHP is dynamiccly typed (no need to declare type).
- Variables make your code flexible and dynamic.
- Good variable names make your code more understandable.

PHP CONSTANTS

Fixed Values, Global Access!

```php
define('SITE_NAME', 'CodeSardar');
const VERSION = '1.0.0';
echo SITE_NAME; // CodeSardar
```

define() Function to create a constant

const Keyword to create a constant

Constant Value cannot be changed

Fixed Value Once defined, it stays the same

Case Sensitive Constant names are case sensitive

Global Accessible anywhere in the script

Configuration Used for settings, paths, keys, etc.

1. WHAT IS A CONSTANT?

A constant is an identifier (name) for a simple value. Once defined, it cannot be changed.

```php
<?php
define('SITE_NAME', 'CodeSardar');
const VERSION = '1.0.0';
echo SITE_NAME;  // CodeSardar
echo VERSION;    // 1.0.0
?>
```

Constants are like variables, except their values cannot be changed after definition.

2. HOW TO CREATE CONSTANTS

Using define() Function

```php
define('NAME', 'Value');
```

- Must use single quotes for the name
- Can be called anywhere after definition

Using const Keyword (PHP 5.3+)

```php
const NAME = 'Value';
```

- Recommended way
- Must be defined in the global scope

3. ACCESSING CONSTANTS

Use the constant name directly.

```php
<?php
define('PI', 3.14159);
echo PI;  // 3.14159
echo PI * 2;  // 6.28318
?>
```

No $ sign is used to access constants. Just use the constant name.

4. CASE SENSITIVITY

Constant names are case sensitive.

```php
<?php
define('SITE_NAME', 'CodeSardar');
echo SITE_NAME;      // Works
echo site_name;      // Error
echo Site_Name;      // Error
?>
```

SITE_NAME, site_name, Site_Name are all different constants.

5. GLOBAL ACCESS

Constants are global and can be used anywhere in the script.

```php
<?php
define('APP_NAME', 'MyApp');

function showName() {
    echo APP_NAME;  // MyApp
}
showName();
?>
```

6. CANNOT BE CHANGED

Trying to change a constant will cause an error.

```php
<?php
define('MAX_LIMIT', 100);
MAX_LIMIT = 200;    // Error!
?>
```

Fatal error: Cannot redeclare constant MAX_LIMIT

7. COMMON USES

- Database configuration
- API keys and tokens
- File paths
- App settings
- Debugging and environment
- Version numbers

8. BEST PRACTICES

- Use UPPERCASE with underscores
- Give meaningful names
- Define constants at the top
- Use const for new projects
- Keep related constants together

QUICK SUMMARY

define() Function to create a constant

const Keyword (PHP 5.3+) to create constant

Fixed Value Cannot be changed after definition

Case Sensitive Constant names are case sensitive

Global Accessible from anywhere in the script

Configuration Best for settings and configuration

TIP

Use constants to make your code secure, maintainable and easy to update!

PHP DATA TYPES

Different Types of Data You Can Use in PHP

```php
<?php
$name = "John";
$age = 25;
$price = 19.99;
$isActive = true;
$items = ["Pen", "Book"];
$user = (object) ["name" => "John"];
$nothing = NULL;
$file = fopen("test.txt", "r");
?>
```

WHAT ARE DATA TYPES?

Data types specify the type of value a variable can hold. PHP supports multiple data types to handle different kinds of data.

DATA TYPES OVERVIEW

String · Integer · Float · Boolean · Array · Object · NULL · Resource

1. STRING

Sequence of characters.

`"Hello, PHP!"`

```php
<?php
$name = "John Doe";
$greeting = 'Hello World';
?>
```

Strings can be in single or double quotes.

2. INTEGER

Whole numbers (no decimal).

`25`

```php
<?php
$age = 25;
$count = -100;
?>
```

No decimals or quotes.

3. FLOAT

Numbers with decimal points.

`19.99`

```php
<?php
$price = 19.99;
$pi = 3.14159;
?>
```

Also called double.

4. BOOLEAN

Represents TRUE or FALSE.

`true / false`

```php
<?php
$isLogin = true;
$isAdmin = false;
?>
```

Used in conditions and comparisons.

5. ARRAY

Holds multiple values in one variable.

`[ "Pen", "Book", "Bag" ]`

```php
<?php
$items = ["Pen", "Book", "Bag"];
$nums = [1, 2, 3, 4];
?>
```

Can be indexed or associative.

6. OBJECT

Instance of a class.

```
{ name: "John",
  age: 25 }
```

```php
<?php
$user = (object) ["name" => "John", "age" => 25];
?>
```

Used in OOP (Object Oriented Programming).

7. NULL

Represents a variable with no value.

`NULL`

```php
<?php
$nothing = NULL;
$name = null;
?>
```

NULL (case-insensitive).

8. RESOURCE

Special variable that holds a resource.

```php
<?php
$file = fopen("test.txt", "r");
$db = mysqli_connect("localhost", "root", "",
"test");
?>
```

Returned by many PHP functions.

QUICK TYPE CHECK FUNCTIONS

Function	Description	Example	Result
is_string()	Checks if value is string	is_string("PHP")	✓
is_int()	Checks if value is integer	is_int(25)	✓
is_float()	Checks if value is float	is_float(19.99)	✓
is_bool()	Checks if value is boolean	is_bool(true)	✓
is_array()	Checks if value is array	is_array([1,2])	✓
is_object()	Checks if value is object	is_object($user)	✓
is_null()	Checks if value is null	is_null(NULL)	✓
is_resource()	Checks if value is resource	is_resource($file)	✓

DATA TYPES ARE DYNAMIC

You don't need to declare the type. PHP automatically converts the type as needed.

EXAMPLE

```php
<?php
$var = "100";       // String
$var = 100;         // Integer
$var = 100.50;      // Float
$var = true;        // Boolean
$var = [1,2,3];     // Array
$var = NULL;        // NULL
?>
```

→ PHP changes the type automatically based on the value.

TIPS

- Choose the right data type for better code performance.
- Use type checking functions to validate data.
- Arrays and Objects are powerful for complex data.

KEY POINTS

- PHP is dynamically typed.
- Different data types serve different purposes.
- Understanding data types is essential for writing efficient PHP.

PHP STRINGS

Work with Text Easily!

```php
<?php
$name = "John";
echo "Hello, $name!";
?>
```

String — Sequence of characters

Single Quotes — Treats everything literally

Double Quotes — Allows interpolation of variables

Concatenation — Join multiple strings together

Escape Characters — Use \\ to escape special characters

String Length — Get length using strlen() function

Text — Strings are used everywhere!

1. WHAT IS A STRING?

A string is a sequence of characters enclosed in quotes.

`"Hello, PHP!"`

```php
<?php
 $msg = "Hello, PHP!";
?>
```

Strings can contain letters, numbers, spaces and special characters.

2. SINGLE QUOTES

Single quotes treat everything literally. Variables will not be parsed.

`'This is $name'`

```php
<?php
 $name = "John";
 echo 'Hello, $name';   // Output: Hello, $name
?>
```

Best for simple text without variables or escape sequences.

3. DOUBLE QUOTES

Double quotes allow variable interpolation and escape sequences.

`"Hello, $name"`

```php
<?php
 $name = "John";
 echo "Hello, $name";   // Output: Hello, John
?>
```

Use when you want to include variables or special characters.

4. INTERPOLATION

Variables inside double quotes are replaced with their values.

```php
<?php
 $name = "John";
 $greet = "Hello, $name!";
 echo $greet;   // Output: Hello, John!
?>
```

Only works with double quotes.

5. CONCATENATION

Use dot (.) operator to join strings.

`"Hello" . " " . "World"`

```php
<?php
$first = "Hello";
$last  = "World";
$full  = $first . " " . $last;
echo $full;   // Output: Hello World
?>
```

Useful when combining multiple strings or variables.

6. ESCAPE CHARACTERS

Use \\ to escape special characters in strings.

Escape	Represents	Example	Output
\n	New Line	"Hello\nWorld"	Hello World
\t	Tab	"Hello\tWorld"	Hello World
\"	Double Quote	"He said \"Hi\""	He said "Hi"
\\	Backslash	"C:\\PHP\\Test"	C:\PHP\Test

Escape sequences help format text.

7. STRING LENGTH

Use strlen() to get the length of a string.

`strlen("Hello") // 5`

```php
<?php
$text = "CodeSardar";
$len = strlen($text);
echo $len;   // Output: 11
?>
```

Counts all characters including spaces and symbols.

8. MORE STRING FUNCTIONS

Useful built-in functions for strings.

Function	Description	Example	Output
strlen()	Length of string	strlen("PHP")	3
strtoupper()	Convert to Uppercase	strtoupper("php")	PHP
strtolower()	Convert to Lowercase	strtolower("PHP")	php
substr()	Extract part of string	substr("Hello", 1, 3)	ell
trim()	Remove whitespace	trim(" Hi ")	"Hi"
str_replace()	Replace text	str_replace("a","@","cat")	c@t

EXAMPLE (ALL TOGETHER)

```php
<?php
 $name = "John";
 $greeting = "Hello, " . $name . "!!\nWelcome to PHP.";
 echo $greeting;
 echo "Length: " . strlen($greeting);   // Output length
?>
```

→

Output:

Hello, John!
Welcome to PHP.
Length: 28

TIPS

- ✔ Use single quotes for plain text.
- ✔ Use double quotes when you need variables.
- ✔ Use concatenation to combine strings.
- ✔ Always escape special characters.

KEY TAKEAWAY

Strings are the foundation of text handling in PHP. Master them to build powerful applications!

PHP STRING FUNCTIONS

Powerful Functions to Work with Text!

```php
<?php
$text = " Hello PHP World ";
echo strtoupper($text);  // HELLO PHP WORLD
?>
```

1. strlen()

Returns the length of a string (number of characters).

`strlen(string)`

```php
<?php
$text = "Hello PHP!";
echo strlen($text);  // 11
?>
```

Output: 11

2. strtoupper()

Converts a string to uppercase.

`strtoupper(string)`

```php
<?php
$text = "Hello PHP!";
echo strtoupper($text); // HELLO PHP!
?>
```

Output: HELLO PHP!

3. strtolower()

Converts a string to lowercase.

`strtolower(string)`

```php
<?php
$text = "Hello PHP!";
echo strtolower($text);  // hello php!
?>
```

Output: hello php!

4. trim()

Removes whitespace or other characters from both sides.

`trim(string)`

```php
<?php
$text = "    Hello PHP!    ";
echo trim($text); // Hello PHP!
?>
```

Output: Hello PHP!

5. str_replace()

Replaces all occurrences of a search string with a replacement string.

`str_replace(search, replace, string)`

```php
<?php
$text = "I love PHP";
echo str_replace("love", "like", $text);
// I like PHP
?>
```

Output: I like PHP

6. substr()

Returns part of a string.

`substr(string, start, length)`

```php
<?php
$text = "Hello PHP World";
echo substr($text, 6, 3); // PHP
?>
```

Output: PHP

7. strpos()

Finds the position of the first occurrence of a string inside another string.

`strpos(string, search)`

```php
<?php
$text = "Hello PHP World";
echo strpos($text, "PHP"); // 6
?>
```

Output: 6

8. explode()

Splits a string by a string (delimiter) into an array.

`explode(separator, string)`

```php
<?php
$text = "PHP,Java,Python";
$arr = explode(",", $text);
print_r($arr);
// Array ( [0] => PHP [1] => Java [2] => Python )
?>
```

Output: `Array ( [0] => PHP [1] => Java [2] => Python )`

QUICK EXAMPLE (ALL TOGETHER)

```php
<?php
$text = "    Hello PHP World    ";
$clean = trim($text);
echo strtoupper($clean);          // HELLO PHP WORLD
echo "\nLength: " . strlen($clean); // Length: 15
echo "\nPosition of 'PHP': " . strpos($clean, "PHP"); // 6
?>
```

COMMON USE CASES

- Validate and clean user input
- Format text (uppercase, lowercase)
- Extract parts of strings
- Search and replace text
- Split strings into arrays

TIPS

- String functions are case-sensitive.
- Indexes start from 0.
- Always check if strpos() returns false (if not found).
- Use trim() to avoid extra spaces.

FUNCTION SUMMARY

Function	What it Does
strlen()	Get the length of a string
strtoupper()	Convert string to UPPERCASE
strtolower()	Convert string to lowercase
trim()	Remove whitespace from both sides
str_replace()	Replace text within a string
substr()	Extract part of a string
strpos()	Find position of first occurrence
explode()	Split string into an array

PHP NUMBERS & MATH

Work with Numbers Like a Pro!

```php
<?php
$a = 16.7;
echo round($a);   // 17
echo sqrt(25);    // 5
?>
```

1. abs()

Returns the absolute value of a number.

`abs(number)`

```php
<?php
$x = -25;
echo abs($x);  // 25
?>
```

Output: 25

2. round()

Rounds a number to the nearest integer or decimals.

`round(number, decimals)`

```php
<?php
$x = 16.789;
echo round($x);       // 17
echo round($x, 2);    // 16.79
?>
```

Output: 17, 16.79

3. ceil()

Rounds a number up to the nearest integer.

`ceil(number)`

```php
<?php
$x = 16.2;
echo ceil($x);  // 17
?>
```

Output: 17

4. floor()

Rounds a number down to the nearest integer.

`floor(number)`

```php
<?php
$x = 16.9;
echo floor($x);   // 16
?>
```

Output: 16

5. sqrt()

Returns the square root of a number.

`sqrt(number)`

```php
<?php
$x = 25;
echo sqrt($x);  // 5
?>
```

Output: 5

6. pow()

Returns the value of x raised to the power of y.

`pow(base, exponent)`

```php
<?php
echo pow(2, 3);   // 8  (2³)
echo pow(5, 2);   // 25 (5²)
?>
```

Output: 8, 25

7. min()

Returns the smallest value from the given numbers.

`min(value1, value2, ...)`

```php
<?php
echo min(10, 5, 20, 3);  // 3
?>
```

Output: 3

8. max()

Returns the largest value from the given numbers.

`max(value1, value2, ...)`

```php
<?php
echo max(10, 5, 20, 3);  // 20
?>
```

Output: 20

QUICK EXAMPLE (ALL TOGETHER)

```php
<?php
$a = -15.78;                          // 15
echo "abs: " . abs($a) . "<br>";      // 15.78
echo "round: " . round($a, 1) . "<br>"; // -15.8
echo "ceil: " . ceil($a) . "<br>";    // -15
echo "floor: " . floor($a) . "<br>";  // -16
echo "sqrt: " . sqrt(81) . "<br>";    // 9
echo "pow: " . pow(3, 3) . "<br>";    // 27
echo "min: " . min(4, 9, 2) . "<br>"; // 2
echo "max: " . max(4, 9, 2) . "<br>"; // 9
?>
```

Output:

abs: 15.78
round: -15.8
ceil: -15
floor: -16
sqrt: 9
pow: 27
min: 2
max: 9

MATH OPERATIONS (BASIC)

Operator	Name	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 3	1.6667
%	Modulus	5 % 3	2

TIPS

- PHP supports integers and floats.
- Math functions work with both.
- Use `round()`, `ceil()`, `floor()` to control decimals.
- `sqrt()` works only with non-negative numbers.
- `min()` and `max()` can take multiple values.

FUNCTION REFERENCE

Function	Description
`abs(x)`	Absolute value
`round(x, n)`	Round to n decimal places (default 0)
`ceil(x)`	Round up to nearest integer
`floor(x)`	Round down to nearest integer
`sqrt(x)`	Square root
`pow(x, y)`	x raised to the power of y
`min(a, b, ...)`	Smallest value
`max(a, b, ...)`	Largest value

PHP OPERATORS

Perform Operations Easily!

```php
<?php
$a = 10;
$b = 5;
echo $a + $b;   // 15
?>
```

+	Addition	−	Subtraction	*	Multiplication	/	Division	%	Modulus	**	Exponent	=	Assignment
	Add two values		Subtract values		Multiply values		Divide values		Remainder of division		Power of a number		Assign value

==	Equal	===	Identical	!	Not Identical
	Check equal (loose)		Check equal (strict)		Check not equal (strict)

1. +

Addition

Adds two values.

`$a + $b`

```php
<?php
$a = 10;
$b = 5;
echo $a + $b;
?>
```

Output: 15

2. -

Subtraction

Subtracts right value from left.

`$a - $b`

```php
<?php
$a = 10;
$b = 5;
echo $a - $b;
?>
```

Output: 5

3. *

Multiplication

Multiplies two values.

`$a * $b`

```php
<?php
$a = 10;
$b = 5;
echo $a * $b;
?>
```

Output: 50

4. /

Division

Divides left value by right.

`$a / $b`

```php
<?php
$a = 10;
$b = 5;
echo $a / $b;
?>
```

Output: 2

5. %

Modulus

Returns the remainder of division.

`$a % $b`

```php
<?php
$a = 10;
$b = 3;
echo $a % $b;
?>
```

Output: 1

6. **

Exponent

Returns the value of left raised to right.

`$a ** $b`

```php
<?php
$a = 2;
$b = 3;
echo $a ** $b;
?>
```

Output: 8

7. =

Assignment

Assigns value to a variable.

`$a = $b`

```php
<?php
$a = 10;
$b = $a;
echo $b;
?>
```

Output: 10

8. ==

Equal (Loose)

Returns true if values are equal.

`$a == $b`

```php
<?php
$a = "10";
$b = 10;
var_dump($a == $b);
?>
```

Output: bool(true)

9. ===

Identical (Strict)

Returns true if values and data types are equal.

`$a === $b`

```php
<?php
$a = "10";
$b = 10;
var_dump($a === $b);
?>
```

Output: bool(false)

10. !=

Not Equal (Loose)

Returns true if values are not equal.

`$a != $b`

```php
<?php
$a = "10";
$b = 20;
var_dump($a != $b);
?>
```

Output: bool(true)

11. !==

Not Identical (Strict)

Returns true if values or data types are not equal.

`$a !== $b`

```php
<?php
$a = "10";
$b = 10;
var_dump($a !== $b);
?>
```

Output: bool(true)

OPERATOR PRECEDENCE (High to Low)

Precedence	Operators
1	**
2	* , / , %
3	+ , -
4	== , === , != , !==
5	=

QUICK TIPS

- Use == to compare values (ignores type).
- Use === to compare values and types (strict).
- Use % to get remainder.
- Use ** for power (PHP 5.6+).
- Parentheses () can change precedence.

MULTIPLE EXAMPLE

```php
<?php
$a = 10; $b = 3;
echo $a + $b;   // 13
echo $a - $b;   // 7
echo $a * $b;   // 30
echo $a / $b;   // 3.3333333333333
echo $a % $b;   // 1
echo $a ** $b;  // 1000
var_dump($a == "10");   // true
var_dump($a === "10");  // false
?>
```

KEY TAKEAWAY

Operators are the building blocks of expressions in PHP. Master them to write powerful and logical code!

NOTE

In PHP, most operators work with both numbers and strings depending on the context.

PHP COMPARISON & LOGICAL OPERATORS

Compare Values & Make Smart Decisions!

```php
<?php
$a = 10;
$b = 20;
if ($a < $b && $b >= 20) {
   echo "Condition is True!";
}
?>
```

COMPARISON OPERATORS
Used to compare two values.
Returns TRUE or FALSE.

LOGICAL OPERATORS
Used to combine multiple conditions.
Returns TRUE or FALSE.

COMPARISON OPERATORS

Operator	Name	Description	Example	Result
>	Greater Than	Returns true if left value is greater than right value.	`<?php echo (10 > 5); ?>`	true
<	Less Than	Returns true if left value is less than right value.	`<?php echo (10 < 5); ?>`	false
>=	Greater Than or Equal	Returns true if left value is greater than or equal to right value.	`<?php echo (10 >= 10); ?>`	true
<=	Less Than or Equal	Returns true if left value is less than or equal to right value.	`<?php echo (10 <= 5); ?>`	false
==	Equal	Returns true if both values are equal.	`<?php echo (10 == "10"); ?>`	true (type conversion)
===	Identical	Returns true if both values are equal and of the same type.	`<?php echo (10 === "10"); ?>`	false (type strict)
!=	Not Equal	Returns true if both values are not equal.	`<?php echo (10 != 5); ?>`	true
!==	Not Identical	Returns true if both values are not equal or not of the same type.	`<?php echo (10 !== "10"); ?>`	true

LOGICAL OPERATORS

Operator	Name	Description	Example	Result
&&	Logical AND	Returns true if both conditions are true.	`<?php $a = true; $b = false; var_dump($a && $b); ?>`	bool(false)
||	Logical OR	Returns true if at least one condition is true.	`<?php $a = true; $b = false; var_dump($a || $b); ?>`	bool(true)
!	Logical NOT	Reverses the result of a condition.	`<?php $a = true; var_dump(!$a); ?>`	bool(false)
AND	AND	Alternative to && (lower precedence)	`<?php $a = true AND $b = false; var_dump($a); ?>`	bool(false)
OR	OR	Alternative to || (lower precedence)	`<?php $a = false OR $b = true; var_dump($a); ?>`	bool(true)
NOT	NOT	Alternative to ! (lower precedence)	`<?php $a = false; var_dump(NOT $a); ?>`	bool(true)

TRUTH TABLE (Logical Operators)

A	B	A && B	A || B
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

NOT TABLE

A	!A
T	F
F	T

T = True
F = False

COMPARISON EXAMPLES

```php
<?php
echo (10 > 5);       // true
echo (10 < 5);       // false
echo (10 >= 10);     // true
echo (10 <= 5);      // false
echo (10 == "10");   // true
echo (10 === "10"); // false
?>
```

LOGICAL EXAMPLES

```php
<?php
$a = 10; $b = 20;
var_dump(($a < $b) && ($b > 15)); // true
var_dump(($$a > $b) || ($b > 15)); // true
var_dump(!($a > $b));             // true
?>
```

TIPS

- ✔ Use === and !== for strict comparison.
- ✔ Logical operators are useful in conditions.
- ✔ Operator precedence matters in complex expressions.

OPERATOR PRECEDENCE (High to Low)

1. ! (NOT)
2. ==, ===, !=, !==, >, <, >=, <=
3. && (AND)
4. || (OR)

NOTE

Use parentheses () to make your expressions clear and avoid confusion.

PHP CONDITIONAL STATEMENTS

Make Decisions in Your Code!

```php
<?php
$score = 75;
if ($score >= 60) {
    echo "You Passed!";
} else {
    echo "Try Again!";
}
?>
```

if — Executes code if condition is true.

else — Executes code if condition is false.

elseif — Checks multiple conditions.

switch — Selects one block from many options.

case — Defines a case in switch statement.

default — Executes if no case matches.

break — Stops execution in switch.

Conditions — Return TRUE or FALSE.

1. if STATEMENT

Executes block of code if the condition is true.

```
if (condition) { ... }
```

```php
<?php
$age = 18;
if ($age >= 18) {
    echo "You are an adult.";
}
?>
```

Runs only when condition is TRUE.

2. if...else STATEMENT

Executes one block if condition is true, otherwise another block.

```
if (condition) { ... } else { ... }
```

```php
<?php
$age = 16;
if ($age >= 18) {
    echo "Adult";
} else {
    echo "Minor";
}
?>
```

One of the blocks will always run.

3. if...elseif...else STATEMENT

Checks multiple conditions sequentially.

```
if (...) { ... } elseif (...) { ... } else { ...}
```

```php
<?php
$marks = 75;
if ($marks >= 90) {
    echo "Grade: A";
} elseif ($marks  >= 75) {
    echo "Grade: B";
} else {
    echo "Grade: C";
}
?>
```

Evaluates conditions from top to bottom.

4. switch STATEMENT

Selects one block of code from many options.

```
switch (expression) { ... }
```

```php
<?php
$day = 3;
switch ($day) {
    case 1:
        echo "Monday";
        break;
    case 2:
        echo "Tuesday";
        break;
    default:
        echo "Other Day";
}
?>
```

Expression is compared with each case.

5. case STATEMENT

Defines a case inside switch statement.

```
case value: ...
```

```php
switch ($color) {
    case "red":
        echo "Red Color";
        break;
    case "blue":
        echo "Blue Color";
        break;
}
```

Case value must be unique.

6. default STATEMENT

Executes if no case matches the expression.

```
default: ...
```

```php
switch ($grade) {
    case 'A':
        echo "Excellent";
        break;
    case 'B':
        echo "Good";
        break;
    default:
        echo "Keep Trying";
}
```

Optional but very useful.

COMMON CONDITIONS

==	Equal	$a == $b
===	Identical	$a === $b
!=	Not Equal	$a != $b
!==	Not Identical	$a !== $b
>	Greater Than	$a > $b
<	Less Than	$a < $b
>=	Greater or Equal	$a >= $b
<=	Less or Equal	$a <= $b
&&	AND	$a > 0 && $b > 0
||	OR	$a > 0 || $b > 0
!	NOT	!($a > 0)

COMPLETE EXAMPLE (ALL TOGETHER)

```php
<?php
$score = 85;
if ($score >= 90) {
    echo "Grade: A";
} elseif ($score >= 75) {
    echo "Grade: B";
} else {
    echo "Grade: C";
}
$day = 2;
switch ($day) {
    case 1: echo "Monday"; break;
    case 2: echo "Tuesday"; break;
    default: echo "Other Day";
}
?>
```

Output:

```
Grade: B
Tuesday
```

Tip: Use indentation and break; to keep your code clean and bug-free.

QUICK REFERENCE

Keyword	Purpose	Example
if	Executes if condition is true	if ($x > 0) { ... }
else	Executes if condition is false	else { ... }
elseif	Checks another condition	elseif ($x == 0) { ... }
switch	Multiple choice selection	switch ($x) { ... }
case	A case in switch	case 1: ...
default	Runs if no case matches	default: ...
break	Stops execution in switch	break;

BEST PRACTICES

- Always use meaningful conditions.
- Use elseif for multiple conditions.
- Always use break; in switch.
- Use default in switch as a safety net.
- Keep your code clean and easy to read.

PHP TERNARY & NULL COALESCING

Write Less Code, Do More!

```php
<?php
$age = 18;
$status = ($age >= 18) ? 'Adult' : 'Minor';
$name = $userName ?? 'Guest';
?>
```

TERNARY OPERATOR

Short way to write IF–ELSE.
Returns one value based on condition.

NULL COALESCING OPERATOR

Returns first write that is set and not NULL.
Useful for default or optional values.

- **Short Condition** — Less code, more efficiency
- **Default Value** — Provide fallback value easily
- **Optional Value** — Handle NULL and undefined
- **Expression** — Works inside expressions

1. TERNARY OPERATOR (? :)

SYNTAX: `condition ? expr_if_true : expr_if_false`

HOW IT WORKS

If condition is true, returns expr_if_true, otherwise returns expr_if_false.

EXAMPLE

```php
<?php
$age = 20;
$result = ($age >= 18) ? 'Adult' : 'Minor';
echo $result;  // Adult
?>
```

MORE EXAMPLES

Example	Output
`$status = ($active) ? 'Yes' : 'No';`	Yes / No
`$grade = ($marks >= 50) ? 'Pass' : 'Fail';`	Pass / Fail
`$msg = ($user) ? 'Welcome ' . $user : 'Guest';`	Welcome John / Guest
`$discount = ($total > 1000) ? 10 : 0;`	10 / 0

NESTED TERNARY

```php
<?php
$marks = 85;
$grade = ($marks >= 80) ? 'A' :
         ($marks >= 60) ? 'B' :
         ($marks >= 40) ? 'C' : 'D';
echo $grade; // A
?>
```

TIP
Use ternary for simple conditions.
For complex logic, use if...else.

2. NULL COALESCING OPERATOR (??)

SYNTAX: `expr1 ?? expr2`

HOW IT WORKS

Returns expr1 if it is set and not NULL, otherwise returns expr2.

EXAMPLE

```php
<?php
$name = $_GET['name'] ?? 'Guest';
echo $name;  // If 'name' not set, prints Guest
$age = $user['age'] ?? 18;
echo $age;  // If not set, prints 18
?>
```

MORE EXAMPLES

Example	Output/Result
`$user = $_GET['user'] ?? 'Guest';`	Guest
`$city = $info['city'] ?? 'Unknown';`	Unknown
`$count = $data['count'] ?? 0;`	0
`$theme = $_COOKIE['theme'] ?? 'light';`	light

CHAINING (Multiple Defaults)

```php
<?php
$name = $_GET['name'] ?? $user['name']
        ?? $config['name'] ?? 'Guest';
echo $name;
?>
```

TIP
Use ?? to provide default values for optional or missing data.

TERNARY vs IF-ELSE

TERNARY	IF-ELSE
✓ Best for simple conditions	✓ Best for complex conditions
✓ Shorter and cleaner code	✓ More readable for large logic
✓ Returns a value	✓ Can execute multiple statements

VS

QUICK REFERENCE

TERNARY (? :)

- condition ? true_value : false_value
- Returns one value
- Use for short if-else

NULL COALESCING (??)

- expr1 ?? expr2
- Returns first non-null value
- Use for default / optional values

BEST PRACTICES

- ✓ Use ?: for simple return values.
- ✓ Use ?? to handle NULL or undefined.
- ✓ Don't overuse ternary for complex logic.
- ✓ Prefer ?? over isset() for default values.

NOTE

Ternary and Null Coalescing operators help you write clean, short and efficient PHP code.

PHP LOOPS

Repeat Tasks, Save Time!

```php
<?php
for ($i = 1; $i <= 5; $i++) {
    echo $i;   // 1 2 3 4 5
}
?>
```

Loop — Repeats a block of code.

Iteration — One complete execution.

break — Stops the loop immediately.

continue — Skips current iteration and continues.

Control — Control how many times to loop.

Loop Types — Different loops for different needs.

Efficiency — Write less code, get more done.

Conditions — Loops run while condition is true.

1. for LOOP

Best for when you know how many times to loop.

```
for (init; condition; increment) { ... }
```

```php
<?php
for ($i = 1; $i <= 5; $i++) {
    echo "Number: $i<br>";
}
?>
```

Used when number of iterations is fixed.

2. while LOOP

Repeats block of code while condition is true.

```
while (condition) { ... }
```

```php
<?php
$i = 1;
while ($i <= 5) {
    echo "Number: $i<br>";
    $i++;
}
?>
```

Condition is checked before loop runs.

3. do...while LOOP

Similar to while loop, but runs at least once.

```
do { ... } while (condition);
```

```php
<?php
$i = 1;
do {
    echo "Number: $i<br>";
    $i++;
} while ($i <= 5);
?>
```

Condition is checked after loop runs.

4. foreach LOOP

Best for arrays. Loops through each element.

```
foreach ($array as $value) { ... }
```

```php
<?php
$colors = ["Red", "Green", "Blue"];
foreach ($colors as $color) {
    echo $color . "<br>";
}
?>
```

Easily loops through arrays and objects.

5. break

Used to terminate the loop immediately.

```php
if ($i == 3) {
    break;
}
```

STOP

6. continue

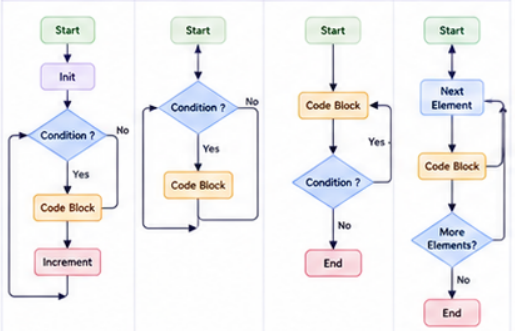
Skips the current iteration and continues with next.

```php
if ($i == 3) {
    continue;
}
```

LOOP TYPES COMPARISON

Loop	Best For	Checks Condition	Runs At Least Once	Use Case
for	Known number of times	Before each iteration	✓	Count-controlled loops
while	Unknown number of times	Before each iteration	✗	Conditional loops
do...while	Unknown number of times	After each iteration	✓	Menu systems, input validation
foreach	Arrays / Objects	Before each iteration	✓	Loop through collection

FLOW OF LOOPS (LOGIC)



PRACTICAL EXAMPLE (ALL TOGETHER)

```php
<?php
$numbers = [10, 20, 30, 40, 50];

echo "Using foreach:<br>";
foreach ($numbers as $num) {
    if ($num == 30) continue;   // Skip 30
    if ($num > 40) break;       // Stop after 40
    echo $num . "<br>";         // 10 20 40
}

echo "<br>Using while:<br>";
$i = 1;
while ($i <= 3) {
    echo "Count: $i<br>";
    $i++;
}
?>
```

TIPS

- ✓ Use for loop when count is known.
- ✓ Use foreach for arrays.
- ✓ Use break and continue wisely.
- ✓ Avoid infinite loops.

BEST PRACTICES

- ✓ Keep conditions simple.
- ✓ Update loop variable correctly.
- ✓ Use meaningful variable names.
- ✓ Test your loops with edge cases.

NOTE

Infinite loops can crash your script or server. Always test loop conditions carefully.

PHP FUNCTIONS

Write Once, Use Anywhere!

```php
<?php
function add($a, $b) {
    return $a + $b;
}
echo add(5, 7);   // Output: 12
?>
```

function — Create a block of reusable code.

Parameters — Variables in function definition.

Arguments — Values passed to function when called.

return — Send value back from function.

Default Parameter — Provide default value for parameter.

Type Declaration — Define expected data type of parameters.

Reusable Code — Write once, use many times.

Function Call — Execute function by its name.

1. FUNCTION SYNTAX

Define a function using function keyword.

```php
function functionName(parameters) {
    // code
    return value; // optional
}
```

```php
<?php
function greet($name) {
    return "Hello, $name!";
}
?>
```

Function names must start with a letter or underscore (_)

2. PARAMETERS & ARGUMENTS

Parameters in definition, arguments in call.

```php
function add($a, $b) {
    return $a + $b;
}
// Function Call
add(10, 20);   // Arguments
```

```php
<?php
function add($a, $b) {
    return $a + $b;
}
echo add(10, 20);  // 30
?>
```

Number of arguments must match parameters.

3. RETURN STATEMENT

return sends value back to the caller.

```php
function square($n) {
    return $n * $n;
}
$result = square(5); // 25
```

```php
<?php
function square($n) {
    return $a * $n;
}
echo square(5);  // 25
?>
```

A function can return only one value.

4. DEFAULT PARAMETER

Set default values for parameters.

```php
function greet($name = "Guest") {
    return "Hello, $name!";
}
greet();       // Hello, Guest
greet("John"); // Hello, John
```

```php
<?php
function greet($name = "Guest") {
    return "Hello, $name!";
}
echo greet();       // Hello, Guest
echo greet("John"); // Hello, John
?>
```

Default value is used if argument is missing.

5. TYPE DECLARATION (PHP 7+)

Declare expected data type.

```php
function divide(int $a, int $b): float {
    return $a / $b;
}

echo divide(10, 2);  // 5
```

```php
<?php
function divide(int $a, int $b): float {
    return $a / $b;
}
echo divide(10, 2);  // 5
?>
```

Helps catch errors early & improves code quality.

6. FUNCTION CALL

Call a function using its name.

```php
function sayHi($name) {
    echo "Hi $name!";
}

sayHi("Sara");  // Function Call
```

```php
<?php
function sayHi($name) {
    echo "Hi $name!";
}
sayHi("Sara");  // Hi Sara!
?>
```

Execution starts from the called function.

EXAMPLE: REAL LIFE USE

Calculate area of rectangle using function.

```php
<?php
function area($length, $width = 1): float {
    return $length * $width;
}
echo area(10, 5);    // 50
echo area(7);        // 7 (width = 1)
?>
```

Output:

```
50
7
```

FUNCTION FLOW

Function Call

```
add(5, 7)
```

→

Function Execution

```
$a = 5
$b = 7
$sum = $a + $b
return $sum;
```

→

Return Value

```
12
```

↓ Returned to Caller

TIPS

- Use meaningful function names.
- Keep functions small and focused.
- Use default parameters for flexibility.
- Use type declarations to avoid errors.
- Always return a value when needed.

COMMON BUILT-IN FUNCTIONS

Function	Purpose	Example
strlen($str)	Length of string	strlen("Hello") // 5
count($arr)	Count elements in array	count([1,2,3]) // 3
date($format)	Current date/time	date("Y-m-d")
is_numeric($val)	Check if value is number	is_numeric(123) // true
round($num)	Round number	round(3.6) // 4

BEST PRACTICES

- Follow naming conventions (e.g. camelCase).
- Avoid too many parameters.
- Document your functions using comments.
- Test your functions regularly.

NOTE

Functions make your code modular, readable and easy to maintain.

PHP ARROW FUNCTIONS

Shorter Code, Modern Style!

```php
<?php
$add = fn($a, $b) => $a + $b;
echo $add(5, 7);   // Output: 12
?>
```

fn() Arrow function syntax.

Arrow Function Compact way to write functions.

Short Syntax Less code, more clarity.

Parameters Accept one or more parameters.

Return Value Automatically returns the expression.

Callback Perfect for callbacks.

Array Functions Great with map, filter, reduce, etc.

Modern PHP Introduced in PHP 7.4+.

1. ARROW FUNCTION SYNTAX

```
fn ( parameters ) => expression
```

Shorter syntax for single-expression functions.
Automatically returns the expression.

Examples:

```php
$square = fn($n) => $n * $n;
$greet  = fn($name) => "Hi $name!";
$sum    = fn($a, $b) => $a + $b;
$ok     = fn() => true;
$pi     = fn() => 3.1415;
```

Best for simple, short, one-line logic.

2. VS NORMAL FUNCTIONS

Normal Function

```php
function sum($a, $b) {
    return $a + $b;
}
```

Arrow Function

```php
$sum = fn($a, $b) => $a + $b;
```

- Less code
- Easier to read
- Automatically returns

3. PARAMETERS

Supports zero, one or multiple parameters.

```php
$noParam    = fn() => "No Param";
$oneParam   = fn($x) => $x * 2;
$multiParam = fn($a, $b) => $a - $b;
```

No need for return keyword.
Use parentheses for multiple parameters.

4. RETURN VALUE

Arrow functions return the expression automatically.

```php
$double = fn($x) => $x * 2;
echo $double(10);   // 20

$message = fn($name) => "Hello $name";
echo $message("Jeet"); // Hello Jeet
```

No return needed.
The result of the expression is returned.

5. CALLBACK EXAMPLE

Great for callbacks in functions.

```php
function process($callback) {
    echo $callback(5);
}
```

```php
process(fn($n) => $n * 3);   // 15
process(fn($n) => $n + 10);  // 15
```

Clean and perfect for callbacks, events, and higher-order functions.

6. ARRAY FUNCTIONS (HIGHLY USEFUL)

Arrow functions work great with array helpers.

`array_map()`	Transform each element
`array_filter()`	Filter elements
`array_reduce()`	Reduce to a single value
`usort()`	Custom sort logic
`array_walk()`	Perform action on each element

```php
$nums = [1, 2, 3, 4, 5];

// MAP: square each number
$squares = array_map(fn($n) => $n * $n, $nums);
// [1, 4, 9, 16, 25]

// FILTER: even numbers
$evens = array_filter($nums, fn($n) => $n % 2 === 0);
// [2, 4]

// REDUCE: sum of all numbers
$sum = array_reduce($nums, fn($carry, $n) => $carry + $n, 0);
// 15
```

7. WHEN TO USE ARROW FUNCTIONS?

- Short, single-line logic
- Callbacks and event handlers
- Array functions (map, filter, reduce, etc.)
- When readability is important
- When you want clean and modern code

TIP
If the function has multiple lines, use normal function instead.

8. QUICK REFERENCE

Task	Arrow Function Example	Result
Double a number	`fn($x) => $x * 2`	`x * 2`
Check even	`fn($n) => $n % 2 === 0`	`true / false`
Return greeting	`fn($name) => "Hi $name"`	Hi Jeet
Add two numbers	`fn($a, $b) => $a + $b`	Sum
No parameter	`fn() => "Done"`	Done

BEST PRACTICES

Use arrow functions for short and simple tasks.

- Keep logic clean and focused.
- Use meaningful parameter names.
- Avoid complex logic in one line.
- Prefer readability over clever code.

NOTE

Arrow functions are single-expression only.
Use normal functions for multi-line logic.

PHP 7.4+

PHP ARRAYS

Store Multiple Values in a Single Variable!

```php
<?php
$fruits = ["Apple", "Banana", "Mango"];
echo $fruits[1];  // Output: Banana
?>
```

[] Square brackets to define arrays.

array() Old way to create arrays.

Indexed Array Numeric keys (0, 1, 2, ...) in order.

Associative Array Named keys with values.

Multidimensional Arrays inside arrays.

Keys Unique identifier for values.

Values Data stored in arrays.

Elements Each value inside an array.

1. INDEXED ARRAY

Numeric index starts from 0.

```php
$arr = [ "Apple", "Banana", "Mango", "Orange" ];
```

```php
<?php
$fruits = ["Apple", "Banana", "Mango", "Orange"];

echo $fruits[0];  // Apple
echo $fruits[2];  // Mango
echo count($fruits); // 4
?>
```

Index	Value
0	Apple
1	Banana
2	Mango
3	Orange

Use numeric index to access values.

2. ASSOCIATIVE ARRAY

Named keys with values.

```php
$arr = [ "name" => "Jeet", "age" => 32, "city" => "Amritsar" ];
```

```php
<?php
$person = [
   "name"   => "Jeet",
   "age"    => 32,
   "city"   => "Amritsar"
];
echo $person["name"];  // Jeet
echo $person["city"];  // Amritsar
?>
```

Key	Value
name	Jeet
age	32
city	Amritsar

Use key name to access values.

3. MULTIDIMENSIONAL ARRAY

Arrays inside arrays.

```php
$arr = [ [1, 2], ["A", "B"], ["x" => 10, "y" => 20] ];
```

```php
<?php
$data = [
   [1, 2],
   ["A", "B"],
   ["x" => 10, "y" => 20]
];
echo $data[0][1];    // 2
echo $data[1][0];    // A
echo $data[2]["y"];  // 20
?>
```

Structure

```
0 → [ 1, 2 ]
1 → [ "A", "B" ]
2 → "x" => 10
    "y" => 20
```

Use multiple indexes or keys to access values.

ARRAY CREATION METHODS

Using [] (Recommended)

```php
$arr = [1, 2, 3];
$arr = ["a" => 1, "b" => 2];
```

Using array() (Old way)

```php
$arr = array(1, 2, 3);
$arr = array("a" => 1, "b" => 2);
```

COMMON ARRAY OPERATIONS

Function / Syntax	Purpose	Example
count($arr)	Total number of elements	count($arr)
in_array($value, $arr)	Check if value exists	in_array(10, $arr)
array_push($arr, value)	Add element at end	array_push($arr,"New")
array_pop($arr)	Remove last element	array_pop($arr)
unset($arr[index/key])	Remove element by index/key	unset($arr[0])

LOOP THROUGH ARRAY

for Loop (Indexed Array)

```php
<?php
$arr = ["Red", "Green", "Blue"];
for ($i = 0; $i < count($arr); $i++) {
   echo $arr[$i] . "<br>";
}
?>
```

foreach Loop (All Arrays)

```php
<?php
$person = ["name" => "Jeet", "age" => 32];
foreach ($person as $key => $value) {
   echo $key . " : " . $value . "<br>";
}
?>
```

QUICK TIPS

- ✔ Indexes start from 0.
- ✔ Keys must be unique.
- ✔ Arrays can store mixed data types.
- ✔ Use foreach for easy looping.

BEST PRACTICES

- ✔ Use meaningful keys.
- ✔ Keep arrays small and focused.
- ✔ Avoid deep nesting.
- ✔ Use modern syntax [].

EXAMPLE: PRACTICAL USE

```php
<?php
$student = [
    "name" => "Jeet",
    "subjects" => ["PHP", "MySQL", "JS"],
    "marks" => [85, 90, 78]
];
echo $student["name"];          // Jeet
echo $student["subjects"][1];   // MySQL
echo $student["marks"][0];      // 85
?>
```

NOTE

- ✔ Always check if key/index exists before accessing.
- ✔ Use isset() to avoid undefined index notice.

```php
isset($arr[0]) ? $arr[0] : "Not Found";
```

- ✔ PHP 7.4+ supports modern array features.

PHP ARRAY FUNCTIONS

Powerful Functions for Easy Array Handling!

```php
<?php
$arr = ["Apple", "Banana", "Mango"];
array_push($arr, "Orange");
echo $arr[3];  // Output: Orange
?>
```

count() Count total elements.

array_push() Add one or more elements at end.

array_pop() Remove last element.

array_shift() Remove first element.

array_unshift() Add one or more elements at start.

array_merge() Merge two or more arrays.

array_slice() Extract a slice from array.

array_search() Search element and return key.

1. count()

Returns the number of elements in an array.

Syntax

```
count(array)
```

```php
<?php
$arr = ["A", "B", "C", "D"];
echo count($arr);  // 4
?>
```

Output: 4

2. array_push()

Adds one or more elements to the end of an array.

Syntax

```
array_push(array, value1, ...)
```

```php
<?php
$arr = ["A", "B"];
array_push($arr, "C", "D");
print_r($arr);
?>
```

Output:

```
Array ( A [1] => B  [2] => C
[3] => D )
```

3. array_pop()

Removes the last element from an array.

Syntax

```
array_pop(array)
```

```php
<?php
$arr = ["A", "B", "C"];
$removed = array_pop($arr);
echo $removed;  // C
print_r($arr);  // [A, B]
?>
```

Output: C

```
Array ( [0] => A  [1] => B )
```

4. array_shift()

Removes the first element from an array.

Syntax

```
array_shift(array)
```

```php
<?php
$arr = ["A", "B", "C"];
$removed = array_shift($arr);
echo $removed;  // A
print_r($arr);  // [B, C]
?>
```

Output: A

```
Array ( [0] => B [1] => C )
```

5. array_unshift()

Adds one or more elements to the beginning of an array.

Syntax

```
array_unshift(array, value1, ...)
```

```php
<?php
$arr = ["B", "C"];
array_unshift($arr, "A");
print_r($arr);
?>
```

Output:

```
Array ( [0] => A [1] => B
[2] => C )
```

6. array_merge()

Merges two or more arrays into one array.

Syntax

```
array_merge(array1, array2, ...)
```

```php
<?php
$a = ["A", "B"];
$b = ["C", "D"];
$c = array_merge($a, $b);
print_r($c);
?>
```

Output:

```
([0] =>  A  [1] => B
[2] => C [3] => D )
```

7. array_slice()

Extracts a slice of the array.

Syntax

```
array_slice(array, start, length)
```

```php
<?php
$arr = ["A", "B", "C", "D", "E"];
$slice = array_slice($arr, 1, 3);
print_r($slice);
?>
```

Output:

```
Array ( [0] => B [1] => C
```

8. array_search()

Searches for an element and returns its key if found.

Syntax

```
array_search(value, array)
```

```php
<?php
$arr = ["A", "B", "C", "D"];
$key = array_search("C", $arr);
echo $key;  // 2
?>
```

Output: 2

REAL LIFE EXAMPLE

Manage a shopping cart using array functions

```php
<?php
$cart = ["Pen", "Pencil"];
array_push($cart, "Notebook");    // Add item
array_unshift($cart, "Eraser");   // Add item at start
array_pop($cart);                 // Remove last item
array_shift($cart);               // Remove first item
print_r($cart);                   // Final cart
?>
```

Output: Array ([0] => Pen [1] => Pencil)

ARRAY FUNCTIONS CHEAT SHEET

Function	Purpose	Example	Result
count()	Count elements	count($arr)	4
array_push()	Add to end	array_push($arr, 'X')	[..., 'X']
array_pop()	Remove last	array_pop($arr)	last element
array_shift()	Remove first	array_shift($arr)	first element
array_unshift()	Add to start	array_unshift($arr, 'X')	['X', ...]
array_merge()	Merge arrays	array_merge($a, $b)	Merged array
array_slice()	Get part of array	array_slice($arr, 1, 2)	Sliced array
array_search()	Search & get key	array_search('C', $arr)	2 (key)

BEST PRACTICES

- Always check if key exists before using it.
- Use count() instead of manual counting.
- Use array_merge() for combining arrays.
- Use array_search() with === for strict search.
- Keep arrays simple and meaningful.

QUICK TIPS

- Arrays in PHP are ordered and can store multiple data.
- Indexes start from 0.
- Use print_r() to debug arrays.
- All these functions return useful results.

NOTE

Some functions change the original array (push, pop, shift, unshift). Use carefully!

PHP ARRAY SORTING

Sort Arrays Easily with Powerful Functions!

```php
<?php
$arr = [3, 1, 4, 1, 5, 9, 2];
sort($arr);
print_r($arr);  // [1, 1, 2, 3, 4, 5, 9]
?>
```

sort() — Sort array in ascending order.

rsort() — Sort array in descending order.

asort() — Sort associative array by value (A → Z).

arsort() — Sort associative array by value (Z → A).

ksort() — Sort associative array by key (A → Z).

krsort() — Sort associative array by key (Z → A).

usort() — Sort array using custom function.

Sorting — Arrange data in a proper order.

1. sort()

Sorts an indexed array in ascending order.

Syntax: `sort(array)`

```php
<?php
$arr = [3, 1, 4, 1, 5, 9, 2];
sort($arr);
print_r($arr);
?>
```

Output:
```
[1, 1, 2, 3, 4, 5, 9]
```

2. rsort()

Sorts an indexed array in descending order.

Syntax: `rsort(array)`

```php
<?php
$arr = [3, 1, 4, 1, 5, 9, 2];
rsort($arr);
print_r($arr);
?>
```

Output:
```
[9, 5, 4, 3, 2, 1, 1]
```

3. asort()

Sorts an associative array by value in ascending order.

Syntax: `asort(array)`

```php
<?php
$arr = ["b"=>2, "a"=>1, "c"=>3];
asort($arr);
print_r($arr);
?>
```

Output:
```
Array ( [a] => 1 [b] => 2 [c] => 3 )
```

4. arsort()

Sorts an associative array by value in descending order.

Syntax: `arsort(array)`

```php
<?php
$arr = ["b"=>2, "a"=>1, "c"=>3];
arsort($arr);
print_r($arr);
?>
```

Output:
```
Array ( [c] => 3 [b] => 2 [a] => 1 )
```

5. ksort()

Sorts an associative array by key in ascending order.

Syntax: `ksort(array)`

```php
<?php
$arr = ["b"=>2, "a"=>1, "c"=>3];
ksort($arr);
print_r($arr);
?>
```

Output:
```
Array ( [a] => 1 [b] => 2 [c] => 3 )
```

6. krsort()

Sorts an associative array by key in descending order.

Syntax: `krsort(array)`

```php
<?php
$arr = ["b"=>2, "a"=>1, "c"=>3];
krsort($arr);
print_r($arr);
?>
```

Output:
```
Array ( [c] => 3 [b] => 2 [a] => 1 )
```

7. usort()

Sorts an indexed array using a custom comparison function.

Syntax: `usort(array, function)`

```php
<?php
$arr = [3, 1, 4, 1, 5];
usort($arr, function($a, $b) {
    return $a <=> $b;  // Ascending
});
print_r($arr);
?>
```

Output:
```
[1, 1, 3, 4, 5]
```

SORTING FUNCTIONS COMPARISON

Function	Type	Sorts By	Order	Maintains Keys	Re-index	Example Use
sort()	Indexed Array	Values	Ascending (Low → High)	✗	✓	Numbers, simple lists
rsort()	Indexed Array	Values	Descending (High → Low)	✗	✓	Top to bottom order
asort()	Associative Array	Values	Ascending (A → Z)	✓	✗	Sort by value, keep keys
arsort()	Associative Array	Values	Descending (Z → A)	✓	✗	Sort by value desc, keep keys
ksort()	Associative Array	Keys	Ascending (A → Z)	✓	✗	Sort by key, keep values
krsort()	Associative Array	Keys	Descending (Z → A)	✓	✗	Sort by key desc, keep values
usort()	Indexed Array	Custom	Custom (Your Rule)	✗	✓	Custom comparison logic

EXAMPLE: CUSTOM SORT USING usort()

```php
<?php
$people = [
    ["name" => "Aman", "age" => 25],
    ["name" => "Ravi", "age" => 20],
    ["name" => "Kiran", "age" => 30]
];

usort($people, function($a, $b) {
    return $a['age'] <=> $b['age'];
});

print_r($people);
?>
```

Output: Sorted by age → 20, 25, 30

BEST PRACTICES

- Use the right function for the right job.
- Remember key preservation in associative arrays.
- Use usort() for complex sorting needs.
- Always check result using print_r().

QUICK TIPS

- sort() & rsort() re-index the array.
- asort(), arsort(), ksort(), krsort() keep the original keys.
- usort() requires a comparison function.

NOTE

Sorting changes the original array.
Make a copy if you need the original data.

PHP DATE & TIME

Work with Dates & Time Like a Pro!

```php
<?php
echo date("Y-m-d H:i:s");    // 2025-05-26 14:35:50
echo time();                 // 1748267750
echo strtotime("tomorrow");  // Timestamp
?>
```

date() — Format a local date and time.

time() — Return current timestamp.

strtotime() — Convert English date to timestamp.

DateTime — Object oriented date & time.

Date Format — Customize date/time output.

Time Zone — Work with different time zones.

Timestamp — Seconds since 1 Jan 1970 (UTC).

Current Date — Get today's date easily.

1. date()

Format a local date and time.

Syntax

```
date(format, timestamp)
```

Format	Example
Y	2025
m	05
d	26
H	14 (24-hour)
i	35 (minutes)
s	50 (seconds)
Y-m-d	2025-05-26
d/m/Y	26/05/2025
H:i:s	14:35:50

```php
<?php
echo date("Y-m-d H:i:s");
echo "<br>";
echo date("d/m/Y");
?>
```

Output:
2025-05-26 14:35:50
26/05/2025

2. time()

Return current Unix timestamp.

Syntax

```
time()
```

```php
<?php
$t = time();
echo $t;
?>
```

Output:
1748267750
(Example Timestamp)

What is Timestamp?

Timestamp is the number of seconds elapsed since 1 January 1970 00:00:00 UTC.

3. strtotime()

Convert English date to timestamp.

Syntax

```
strtotime(datetime, now)
```

```php
<?php
$t1 = strtotime("2025-05-26");
$t2 = strtotime("tomorrow");
$t3 = strtotime("+7 days");
echo $t1 . "<br>";
echo $t2 . "<br>";
echo $t3;
?>
```

Output:
1748217600
1748354150
1748862950
(Example Timestamps)

4. DateTime (OOP)

Date & Time with OOP.

Syntax

```
new DateTime(timezone)
```

```php
<?php
$dt = new DateTime("now",
  new DateTimeZoze("Asia/Kolkata");

echo $dt->format("Y-m-d H:i:s");
?>
```

Output:
2025-05-26 14:35:50

Useful Methods

- format() – Format date/time
- modify() – Modify date/time
- add() – Add interval
- sub() – Subtract interval

5. Date Format

Customize output using format.

Example

```
echo date("d-m-Y H:i:s");
```

```php
<?php
$cho date("d-m-Y H:i:s");
echo "<br>";
echo date("l, jS F Y");
?>
```

Output:
26-05-0-26 14:35:50
Monday, 26th May 2025

6. Time Zone

Work with different time zones.

Example

```
date_default_timezone_set("America/New_York");
echo date("Y-m-d H:i:s T");
```

```php
<?php
date_default_timezone_set("Asia/Kolkata");
echo date("Y-m-d H:i:s T");
?>
```

Output:
2025-05-26 14:35:50 IST

Popular Time Zones

Asia/Kolkata
America/New_York
Europe/London
Australia/Sydney

7. Current Date

Get current date easily.

Example

```
date("Y-m-d");
```

```php
<?php
echo date("Y-m-d");
echo "<br>";
echo date("l");
?>
```

Output:
2025-05-26
Monday

COMMON DATE OPERATIONS

Task	Function / Code	Example
Current Date & Time	date("Y-m-d H:i:s")	2025-05-26 14:35:50
Current Date	date("Y-m-d")	2025-05-26
Add 10 Days	strtotime("+10 days")	Timestamp
Yesterday	strtotime("yesterday")	Timestamp
Next Month	strtotime("next month")	Timestamp
First Day of Month	date("Y-m-01")	2025-05-01
Last Day of Month	date("Y-m-t")	2025-05-31

DATE DIFFERENCE

Find the difference between two dates.

```php
<?php
$d1 = new DateTime("2025-05-01");
$d2 = new DateTime("2025-05-26");
$diff = $d1->diff($d2);
echo $diff->format("%a days"); // Total days
?>
```

Output: 25 days

PRACTICAL EXAMPLE

Next Birthday Countdown

```php
<?php
$birthday = new DateTime("15 August 2025");
$today = new DateTime();
$diff = $today->diff{birthday, false);
echo "Next Birthday in: " . $diff->format("%a days");
?>
```

Output: Next Birthday in: 80 days (Example)

BEST PRACTICES

- Always use meaningful date formats.
- Set timezone using date_default_timezone_set().
- Use DateTime for complex calculations.
- Store dates in database in Y-m-d H:i:s format.

NOTE

Timestamps are in UTC by default.
Timezone changes output, not timestamp.

QUICK TIPS

- Use date() for simple formatting.
- Use strtotime() for text to date conversion.
- Remember timestamp is in seconds.
- Use DateTime for advanced date handling.

PHP FILE HANDLING

Work with Files: Open, Read, Write & More!

```php
<?php
$text = file_get_contents("note.txt");
echo $text;
// Read file content
?>
```

Files — Store data permanently.

Read — Get data from files.

Write — Write data to files.

Functions — Built-in functions make it easy.

Safe — Always close files properly.

Text/Binary — Work with text or binary files.

Permissions — Check file permissions before using.

Exists — Check if file exists or not.

1. fopen()

Opens a file or URL.

Syntax

```
fopen($filename, $mode)
```

```php
<?php
$file = fopen("demo.txt", "r");
if ($file) {
  echo "File opened successfully!";
} else {
  echo "Unable to open file!";
}
?>
```

Common Modes:

- `r` - Read only
- `w` - Write only (truncate)
- `a` - Append
- `r+` - Read/Write
- `a+` - Read/Append

2. fclose()

Closes an open file.

Syntax

```
fclose($file)
```

```php
<?php
$file = fopen("demo.txt", "r");
// some operations
fclose($file);

echo "File closed.";
?>
```

Note: Always close the file after operations to free resources.

3. fread()

Reads data from a file.

Syntax

```
fread($file, $length)
```

```php
<?php
$file = fopen("demo.txt", "r");
$data = fread($file, 20);
echo $data;
fclose($file);
?>
```

Output (first 20 bytes):

```
Hello, this is a PHP
```

4. fwrite()

Writes data to a file.

Syntax

```
fwrite($file, $string)
```

```php
<?php
$file = fopen("demo.txt", "w");
$written = fwrite($file, "Hello
PHP File!");
echo $written . " bytes written";
fclose($file);
?>
```

Output:

```
12 bytes written
```

5. file_get_contents()

Reads entire file into a string.

Syntax

```
file_get_contents($filename)
```

```php
<?php
$data = file_get_contents("demo.txt");
echo $data;
?>
```

Advantages:

- ✔ Easy to use
- ✔ Reads whole file at once
- ✔ Returns string

6. file_put_contents()

Writes data to a file.

Syntax

```
file_put_contents($filename, $data)
```

```php
<?php
$data = "This is written using
file_put_contents().";
file_put_contents("demo.txt", $data);
echo "Data saved!";
?>
```

Note:

Creates file if it doesn't exist.
Overwrites file by default.

7. Other Useful Functions

- **file_exists($file)** — Check if file exists.
- **unlink($file)** — Delete a file.
- **filesize($file)** — Get file size in bytes.
- **filemtime($file)** — Get last modified time.
- **is_readable($file)** — Check if file is readable.
- **is_writable($file)** — Check if file is writable.

EXAMPLE: READ & WRITE FILE

```php
<?php
$file = fopen("example.txt", "w");          // Open for writing
fwrite($file, "Hello PHP File Handling!");  // Write data
fclose($file);                              // Close file

$file = fopen("example.txt", "r");          // Open for reading
(content = fread($file, filesize("example.txt")); // Read data
fclose($file);                              // Close file

echo $content;                              // Output
?>
```

```
Output: Hello PHP File Handling!
```

FILE MODES CHEAT SHEET

Mode	Description	Pointer Start	If File Not Exists	If File Exists
r	Read only	Beginning	Error	Open
w	Write only	Beginning	Create	Truncate (erase)
a	Append only	End	Create	Append
r+	Read/Write	Beginning	Error	Open
w+	Read/Write	Beginning	Create	Truncate
a+	Read/Append	End	Create	Append

BEST PRACTICES

- ✔ Always check if file opened successfully.
- ✔ Close files using fclose().
- ✔ Use appropriate mode for performance.
- ✔ Validate and sanitize file paths.
- ✔ Handle exceptions and errors properly.
- ✔ Set proper file permissions.

QUICK TIPS

- Use file_get_contents() for quick read.
- Use file_put_contents() for quick write.
- Use fread()/fwrite() for large files.
- Always close the file.
- Check file permissions if issues occur.

NOTE

File handling functions work on the server.
Be careful with file paths and permissions!

PHP INCLUDE & REQUIRE

Include External Files and Reuse Code Easily!

```php
<?php
include 'config.php';
require_once 'db.php';
echo "Files Included Successfully!";
?>
```

Reusable Files — Write once, use anywhere.

Cleaner Code — Keep code organized.

Error Handling — Require stops script on error.

Once is Best — Prevents multiple inclusions.

Header / Footer — Use common header and footer files.

Configuration — Store settings in separate files.

Easier Maintenance — Update one file, affects all.

1. include

Includes a file and continues execution even if file is missing.

Syntax

```php
include 'file.php';
```

```php
<?php
include 'header.php';
echo "Page Content";
include 'footer.php';
?>
```

Note: If the file is missing, a warning is shown but the script continues.

2. include_once

Includes a file only once. Prevents multiple inclusion.

Syntax

```php
include_once 'file.php';
```

```php
<?php
include_once 'config.php';
include_once 'config.php';
echo "Included Once!";
?>
```

Note: If the file is included again, it will not be included a second time.

3. require

Includes a file and stops execution if file is missing.

Syntax

```php
require 'file.php';
```

```php
<?php
require 'db.php';
echo "After DB Connection";
?>
```

Note: If the file is missing, a fatal error occurs and script stops.

4. require_once

Includes a file only once and stops execution if file is missing.

Syntax

```php
require_once 'file.php';
```

```php
<?php
require_once 'functions.php';
require_once 'functions.php';
echo "Required Once!";
?>
```

Note: Best practice for required files. Combines both features.

5. Common Use Cases

Header File
`include 'header.php';`
(Logo, Menu, CSS links)

Footer File
`include 'footer.php';`
(Footer, JS links, Copyright)

Configuration File
`require_once 'config.php';`
(Database, Site Settings)

Function Library
`require_once 'functions.php';`
(Custom Functions)

BEST PRACTICES

- Use require_once for important files (config, DB).
- Use include_once for optional files.
- Keep common files in separate folder.
- Use relative paths for better portability.

EXAMPLE PROJECT STRUCTURE

```
myproject/
├── header.php      (Common Header)
├── footer.php      (Common Footer)
├── config.php      (Configurations)
├── functions.php   (Custom Functions)
└── index.php       (Main Page)
```

EXAMPLE: USING INCLUDE & REQUIRE

```php
<?php
include 'config.php';   // Load configuration
include 'header.php';   // Load header

echo "<h1>Welcome to My Website</h1>";

include 'footer.php';   // Load footer
?>
```

INCLUDE vs REQUIRE

Feature	include / include_once	require / require_once
File Missing	Warning (script continues)	Fatal Error (script stops)
Execution	Continues	Stops
Use Case	Optional files	Important files
Best For	Non-critical content	Core functionality
Once Version	include_once	require_once

QUICK TIP

Use __DIR__ for perfect paths:

```php
require_once __DIR__ . '/config/config.php';
```

NOTE

Always check file paths carefully to avoid errors and warnings.

PHP FORMS

Create, Handle & Process Form Data Easily!

```php
<?php
if($_SERVER["REQUEST_METHOD"] == "POST"){
    echo "Form Submitted Successfully!";
}
?>
```

<form> Defines an HTML form in PHP.

method GET or POST – how data is sent.

action Script URL where data is sent.

input Create input fields (text, email, etc.).

textarea Multi-line text input field.

select Dropdown list of options.

button Clickable button (submit/reset).

Form Data User data collected from the form.

Submit Send form data to the server.

1. FORM STRUCTURE & SYNTAX

Basic Form Syntax

```html
<form action="process.php" method="POST">
    <!-- Form Elements -->
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
```

Method Types

Method	Description	Use Case
GET	Data shows in URL	Search, Filters
POST	Data hidden in request	Login, Forms

Note:

- Use POST for sensitive data (password, email, etc.)
- Always validate and sanitize user input.

2. EXAMPLE PHP FORM

Name: Enter your name

Email: Enter your email

Message: Type your message...

Country: Select Country

Gender: Male Female Other

Subscribe: Yes, send me updates

Submit Reset

This form sends data to process.php using POST method.

3. PROCESS FORM DATA (process.php)

```php
<?php
if($_SERVER["REQUEST_METHOD"] == "POST"){
    $name      = $_POST['name'];
    $email     = $_POST['email'];
    $message   = $_POST['message'];
    $country   = $_POST['country'];
    $gender    = $_POST['gender'];
    $subscribe = isset($_POST['subscribe'])
                 ? $_POST['subscribe'] : 'No';

    // Display the submitted data
    echo "<h3>Form Submitted Successfully!</h3>";
    echo "<ul>";
    echo "<li><b>Name:</b> $name</li>";
    echo "<li><b>Email:</b> $email</li>";
    echo "<li><b>Message:</b> $message</li>";
    echo "<li><b>Country:</b> $country</li>";
    echo "<li><b>Gender:</b> $gender</li>";
    echo "<li><b>Subscribe:</b> $subscribe</li>";
    echo "</ul>";
} else {
    echo "No data received!";
}
?>
```

4. SAMPLE OUTPUT

Form Submitted Successfully!

- **Name:** Lovejeet Singh
- **Email:** lovejeet@example.com
- **Message:** Hello! This is a test message.
- **Country:** India
- **Gender:** Male
- **Subscribe:** Yes

5. COMMON INPUT TYPES

Type	Code Example	Description
text	`<input type="text" name="name">`	Single line text
email	`<input type="email" name="email">`	Email field with validation
password	`<input type="password" name="pass">`	Password field
number	`<input type="number" name="age">`	Numeric input
date	`<input type="date" name="dob">`	Date picker
checkbox	`<input type="checkbox" name="agree">`	Checkbox option
radio	`<input type="radio" name="gender">`	Radio button option
file	`<input type="file" name="file">`	File upload
hidden	`<input type="hidden" name="token">`	Hidden data

6. TEXTAREA & SELECT EXAMPLES

Textarea

```html
<textarea name="message" rows="4" cols="30"
placeholder="Type your message..."></textarea>
```

Select Dropdown

```html
<select name="country">
  <option value="">Select Country</option>
  <option value="india">India</option>
  <option value="usa">USA</option>
  <option value="uk">UK</option>
</select>
```

7. BEST PRACTICES

- Use POST method for sensitive data.
- Validate and sanitize all user inputs.
- Always use `hmtlspeilachars()` to prevent XSS.
- Use proper labels for accessibility.
- Provide meaningful error messages.
- Use `enctype="multipart/form-data"` for file upload.

QUICK TIPS

- `name` attribute is required for all inputs.
- `action` is the file that processes the form.
- `method="post"` is more secure than GET.
- Use placeholder for better UX.
- Always test your form after changes.

IMPORTANT NOTE

Never trust user input.
Always validate, sanitize, and secure your form data before using it.

PHP GET & POST

Handle Form Data with GET & POST Methods!

```php
<?php
if(isset($_POST['submit'])){
    echo "Form Submitted!";
}
?>
```

GET Request — Sends data in URL as query string.

POST Request — Sends data in HTTP request body.

Query String — Data appended to URL after ?

Form Data — Data sent in the body (hidden).

Request Method — Tells server how data is being sent.

Submit — Send form data to the server.

GET REQUEST (Using $_GET)

GET method appends data to the URL.
Visible in browser address bar.
Best for non-sensitive data.

Example URL:

page.php?name=Lovejeet&city=Amritsar

Access Data with $_GET

```php
<?php
$name = $_GET['name'];
$city = $_GET['city'];
echo "Name: $name <br>";
echo "City: $city";
?>
```

VS

POST REQUEST (Using $_POST)

POST method sends data in the request body.
Not visible in URL.
Best for sensitive or large data.

Example (Form Submit):

```html
<form action="process.php" method="POST">
  <input type="text" name="name" placeholder="Enter Name">
  <input type="email" name="email" placeholder="Enter Email">
  <button type="submit" name="submit">Submit</button>
</form>
```

Access Data with $_POST

```php
<?php
$name  = $_POST['name'];
$email = $_POST['email'];
if(isset($_POST['submit'])){
    echo "Name: $name <br>";
    echo "Email: $email";
}
?>
```

GET Request Flow

Browser → page.php?name=Lovejeet&city=Amritsar (Query String in URL) → Server

GET in Browser (Visible in URL)

https://example.com/page.php?name=Lovejeet&city=Amritsar

POST Request Flow

Browser → Form Data in Request Body → Server

POST in Browser (Not Visible in URL)

https://example.com/process.php

WHEN TO USE?

Use GET When:

- Data is non-sensitive
- Data is small in size
- You want to bookmark the URL
- Example: Search, filters, pagination

Use POST When:

- Data is sensitive (password, email, etc.)
- Data is large
- You don't want data in URL
- Example: Login, Register, Form Submit

DIFFERENCE BETWEEN GET & POST

Feature	GET	POST
Data Location	URL (Query String)	Request Body
Visibility	Visible in URL	Not Visible in URL
Data Size	Limited (~2048 characters)	No Limit
Security	Less Secure	More Secure
Bookmark	Can be bookmarked	Cannot be bookmarked
Caching	Can be cached	Not cached
Use Case	Search, Filter, Pagination	Login, Register, Form Submit

COMPLETE EXAMPLE

form.html (HTML Form)

```html
<form action="process.php" method="POST">
  <input type="text" name="name" placeholder="Name"><br><br>
  <input type="email" name="email" placeholder="Email"><br><br>
  <button type="submit" name="submit">Submit</button>
</form>
```

process.php (Handle Form Data)

```php
<?php
if(isset($_POST['submit'])){
    $name  = $_POST['name'];
    $email = $_POST['email'];
    echo "<h3>Form Submitted Successfully!</h3>";
    echo "Name: $name <br>";
    echo "Email: $email";
} else {
    echo "No data received!";
}
?>
```

BEST PRACTICES

- Always validate and sanitize user input.
- Use POST for sensitive data.
- Use HTTPS for secure data transfer.
- Never trust user input.

IMPORTANT NOTE

Never store sensitive data in URL.
Always use POST for login, registration and important form submissions.

PHP FORM VALIDATION

Validate, Sanitize & Secure User Input!

```php
<?php
if ($errors) {
    echo "Please fix the errors below!";
} else {
    echo "Form Submitted Successfully!";
}
?>
```

required — Field must not be empty.

filter_var() — Filter & validate data.

FILTER_VALIDATE_EMAIL — Validate email format.

trim() — Remove spaces from start & end.

htmlspecialchars() — Convert special characters to HTML.

Input Validation — Check user input before processing.

Sanitize — Clean input to prevent XSS & attacks.

Error Message — Show user-friendly error messages.

1. BASIC VALIDATION RULES

Rule	Description
required	Field must not be empty.
email	Must be a valid email address.
min_length	Minimum number of characters.
max_length	Maximum number of characters.
numeric	Must contain only numbers.
alpha	Must contain only letters.
alpha_numeric	Letters and numbers only.
url	Must be a valid URL.

Tip: Always validate on server side. Client side validation can be bypassed.

2. EXAMPLE VALIDATION SCRIPT

```php
<?php
$errors = [];
$name = trim($_POST['name'] ?? '');
$maail = trim($_POST['email'] ?? '');
$age = trim($_POST['age'] ?? '');

// Name validation
if (empty($name)) {
    $errors['name'] = "Name is required.";
} elseif (strlen($name) < 3) {
    $errors['name'] = "Name must be at least 3 characters.";
}

// Email validation
if (empty($email)) {
    $errors['email'] = "Email is required.";
} elseif (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $errors['email'] = "Enter a valid email address.";
 }
?>
```

3. SANITIZE USER INPUT

Function	Purpose
trim()	Remove spaces from start & end
stripslashes()	Remove backslashes
htmlspecialchars()	Convert HTML special chars
filter_var()	Filter & validate data
intval()	Convert to integer
floatval()	Convert to float
strip_tags()	Remove HTML & PHP tags

Example: Sanitize Input

```php
<?php
$name = trim($_POST['name']);
$name = htmlspecialchars($name, ENT_QUOTES, 'UTF-8');
$age = filter_var($_POST['age'], FILTER_VALIDATE_INT);
?>
```

4. SAMPLE FORM WITH VALIDATION

```php
<form action="validate.php" method="POST" novalidate>
  <div>
    <label>Name:</label><br>
    <input type="text" name="name">
    <span class="error"><?= $errors['name'] ?? '' ?></span>
  </div><br>
  <div>
    <label>Email:</label><br>
    <input type="email" name="email">
    <span class="error"><?= $errors['email'] ?? '' ?></span>
  </div><br>
  <div>
    <label>Age:</label><br>
    <input type="number" name="age">
    <span class="error"><?= $errors['age'] ?? '' ?></span>
  </div><br>
  <button type="submit">Submit</button>
</form>
```

CSS for Error Message

```css
.error {
  color: red;
  font-size: 14px;
  margin-top: 5px;
}
```

5. COMMON filter_var() EXAMPLES

Validation	Example
FILTER_VALIDATE_EMAIL	filter_var($email, FILTER_VALIDATE_EMAIL)
FILTER_VALIDATE_URL	filter_var($url, FILTER_VALIDATE_URL)
FILTER_VALIDATE_INT	filter_var($age, FILTER_VALIDATE_INT)
FILTER_VALIDATE_FLOAT	filter_var($price, FILTER_VALIDATE_FLOAT)
FILTER_SANITIZE_STRING	filter_var($str, FILTER_SANITIZE_STRING)
FILTER_SANITIZE_EMAIL	filter_var($email, FILTER_SANITIZE_EMAIL)

6. ERROR HANDLING EXAMPLE

```php
<?php if (!empty($errors)): ?>
  <div class="error-box">
    <h3>Please fix the following errors:</h3>
    <ul>
      <?php foreach ($errors as $error): ?>
        <li><?= $error ?></li>
      <?php endforeach; ?>
    </ul>
  </div>
<?php endif; ?>
```

Result on Screen

Please fix the following errors:

- Name is required.
- Enter a valid email address.
- Age must be a number.

Name: Name is required.

Email: Enter a valid email address.

Age: Age must be a number.

Submit

7. BEST PRACTICES

- Always validate and sanitize all user input.
- Use filter_var() for validation.
- Show clear and helpful error messages.
- Never trust user input.
- Use prepared statements for database queries.

8. IMPORTANT NOTE

Client-side validation improves user experience, but server-side validation is essential for security.

PHP SUPERGLOBALS

Built-in Variables available in entire script scope

```php
<?php
echo $_SERVER['PHP_SELF'];
echo $_GET['id'];
echo $_SESSION['username'];
?>
```

$_GET

Contains variables passed via GET method.

$_POST

Contains variables passed via POST method.

$_REQUEST

Contains variables from GET, POST and COOKIE.

$_SERVER

Contains information about headers, paths and script locations.

$_SESSION

Contains session variables registered to a session.

$_COOKIE

Contains variables passed via HTTP cookies.

$_FILES

Contains variables related to files uploaded via form.

$_ENV

Contains variables from environment settings.

$_GET

Used to collect data after submitting an HTML form with GET method.

Example:

```php
<?php
$name = $_GET['name'];
$id = $_GET['id'];
?>
```

Sample URL:

```
page.php?name=Lovejeet&id=25
```

Result:

```
name = Lovejeet
id = 25
```

$_POST

Used to collect data after submitting an HTML form with POST method.

Example:

```php
<?php
$email = $_POST['email'];
$age = $_POST['age'];
?>
```

Form Method:

```
<form method="POST">..</form>
```

Result:

```
email = user@example.com
age = 20
```

$_REQUEST

Contains data from $_GET, $_POST and $_COOKIE.

Example:

```php
<?php
$q = $_REQUEST['q'];
?>
```

Can come from:

GET
POST
COOKIE

Result:

```
q = search text
```

$_SERVER

Contains server and execution environment information.

Example:

```php
<?php
$host = $_SERVER['HTTP_HOST'];
$php = $_SERVER['PHP_VERSION'];
?>
```

Common Keys:

PHP_SELF, SERVER_NAME, HTTP_HOST, REQUEST_URI, SCRIPT_NAME, REMOTE_ADDR and more...

Result (Example):

```
host = localhost
php = 8.2.7
```

$_SESSION

Stores session data across multiple pages.

Example:

```php
<?php
session_start();
$_SESSION['user'] = 'Jeet';
echo $_SESSION['user'];
?>
```

Note:

Session must be started with session_start().

Result:

```
user = Jeet
```

$_COOKIE

Stores data on user's browser in the form of cookies.

Example:

```php
<?php
$theme = $_COOKIE['theme'];
?>
```

Set Cookie:

```
setcookie("theme", "dark", time() + 86400);
```

Result:

```
theme = dark
```

$_FILES

Handles data uploaded through an HTML form (file upload).

Example:

```php
<?php
$fileName = $_FILES['file']['name'];
?>
```

Common Keys:

name, type, tmp_name, error, size

Result (Example):

```
fileName = photo.jpg
size = 204800
```

$_ENV

Contains environment variables from the server.

Example:

```php
<?php
$env = $_ENV['PATH'];
?>
```

Common Use:

Server configuration, API keys, system paths and more.

Result (Example):

```
PATH = /usr/local/bin:/usr/bin
```

QUICK EXAMPLE (All Superglobals)

```php
<?php
echo "GET Name: " . ($_GET['name'] ?? '');
echo "<br>POST Email: " . ($_POST['email'] ?? '');
echo "<br>Request Q: " . ($_REQUEST['q'] ?? '');
echo "<br>Server Host: " . $_SERVER['HTTP_HOST'];
echo "<br>Session User: " . ($_SESSION['user'] ?? "Guest");
echo "<br>Cookie Theme: " . ($_COOKIE['theme'] ?? "Not Set");
echo "<br>Uploaded File: " . ($_FILES['file']['name'] ?? "No File");
echo "<br>Env Path: " . ($_ENV['PATH'] ?? '');
?>
```

SUPERGLOBALS OVERVIEW TABLE

Superglobal	Source	Data Type	Scope	Writable
$_GET	URL Query String	Array	Global	No
$_POST	Form Data (POST)	Array	Global	No
$_REQUEST	GET + POST + COOKIE	Array	Global	No
$_SERVER	Server & Execution Info	Array	Global	No
$_SESSION	Session Data	Array	Global	No
$_COOKIE	Cookies	Array	Global	No
$_FILES	Uploaded Files	Array	Global	No
$_ENV	Environment Variables	Array	Global	No

BEST PRACTICES

- Always validate and sanitize user input.
- Use hnmlspocialchars() to prevent XSS.
- Use filter_var() for filtering and validation.
- Never trust user input.
- Use prepared statements for DB queries.
- Keep session data minimal and secure.

QUICK TIPS

- Use GET for search, filters, pagination.
- Use POST for login, forms, sensitive data.
- Always check if key exists using isset().
- Never expose sensitive data in URL.

IMPORTANT NOTE

Never store passwords, sensitive data or personal information in GET, COOKIE or visible URLs.

PHP SESSIONS

Store User Data Across Multiple Pages!

```php
<?php
session_start();
$_SESSION['user'] = 'Lovejeet';
echo $_SESSION['user'];
?>
```

session_start() — Start or resume a session.

$_SESSION — Superglobal used to store session data.

Set Session — Store data in session variables.

Read Session — Access session data anytime.

Update Session — Change session values easily.

Unset — Remove specific session variables.

Destroy — End session and clear all session data.

Login State — Keep user login state across pages.

1. START SESSION

Always start session at the top of your PHP file.

```php
<?php
// Start or resume a session
session_start();
?>
```

Note: session_start() must be called before any output (HTML).

2. SET SESSION

Store data in session variables using $_SESSION.

```php
<?php
session_start();
$_SESSION['username'] = 'Jeet';
$_SESSION['role'] = 'Admin';
$_SESSION['login'] = true;
?>
```

This data will be available on other pages.

3. READ SESSION

Access session data using $_SESSION.

```php
<?php
session_start();
echo "User: " . $_SESSION['username'];
echo "Role: " . $_SESSION['role'];
?>
```

Always check if variable is set before using it.

```php
<?php
if(isset($_SESSION['username'])) {
    echo $_SESSION['username'];
}
?>
```

4. UPDATE SESSION

Update existing session values anytime.

```php
<?php
session_start();
$_SESSION['username'] = 'Lovejeet';
$_SESSION['last_login'] = time();
?>
```

Just assign new value to update the session variable.

5. UNSET SESSION

Remove specific session variable using unset().

```php
<?php
session_start();
unset($_SESSION['role']);
unset($_SESSION['last_login']);
?>
```

Only removed variable will be deleted, session continues.

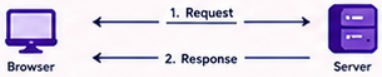
6. DESTROY SESSION

End session and clear all session data.

```php
<?php
session_start();
session_unset();  // Clear all vars
session_destroy(); // Destroy session
?>
```

After destroy, session data will not be available.

HOW SESSIONS WORK?



1. User makes request.
2. Server creates session and stores data.
3. Server sends session ID to browser (via cookie).
4. Browser sends session ID with next requests.
5. Server uses session ID to get user data.

SESSION STORAGE

By default, sessions are stored on the server in the following directory:

/tmp (Linux) or C:\Windows\Temp (Windows)

You can change session.save_path in php.ini

7. LOGIN STATE EXAMPLE

Simple login system using sessions.

login.php

```php
<?php
session_start();
if($_SERVER['REQUEST_METHOD'] == 'POST'){
    $user = $_POST['username'];
    $pass = $_POST['password'];
    // Check user (example only)
    if($user == 'jeet' && $pass == '1234'){
      $_SESSION['username'] = $user;
      $_SESSION['login'] = true;
      header('Location: dashboard.php');
      exit();
    } else {
      $error = 'Invalid Credentials!';
    }
}
?>
```

dashboard.php (Protected Page)

```php
<?php
session_start();
if(!isset($_SESSION['login'])){
    header('Location: login.php');
    exit();
}
echo "Welcome " . $_SESSION['username'];
echo "<br>Login State: Active";
?>
```

Explanation:
If user is not logged in, redirect to login page.
If logged in, show dashboard content.

CHECK LOGIN STATUS

Use this code to check if user is logged in or not on any page.

```php
<?php
session_start();
if(isset($_SESSION['login']) &&
        $_SESSION['login'] == true){
    echo "Logged In";
} else {
    echo "Not Logged In";
}
?>
```

TIPS

- Always start session at the top.
- Keep session data minimal.
- Destroy session on logout.
- Use HTTPS for secure sessions.

COMMON FUNCTIONS

Function	Purpose
session_start()	Start or resume a session
session_id()	Get current session ID
session_unset()	Remove all session variables
session_destroy()	Destroy the session
isset($_SESSION['key'])	Check if session variable is set

BEST PRACTICES

- Regenerate session ID after login.
- Validate and sanitize user input.
- Set session cookie parameters (httponly, secure).
- Logout and destroy session on exit.

NOTE

Sessions are stored on the server, not in the browser.
Do not store sensitive data in sessions.
Always logout the user properly.

PHP COOKIES

Store Small Data in User's Browser!

```php
<?php
setcookie("username", "Lovejeet", time() + 86400, "/");
echo $_COOKIE['username'];
?>
```

setcookie() — Function used to create a cookie.

$_COOKIE — Superglobal used to access cookies.

Cookie Name — Unique name to identify cookie.

Value — Data stored in the cookie.

Expiry — Time (in seconds) until cookie expires.

Path — Specifies the path cookie is available.

Secure — Send cookie only over HTTPS.

HttpOnly — Protects cookie from JavaScript access.

1. setcookie() SYNTAX

```
setcookie(name, value, expiry, path, domain, secure, httponly);
```

Parameter	Required	Description	Default
name	Yes	Name of the cookie	—
value	Yes	Value to be stored	—
expiry	No	Expiry time (timestamp)	0 (Session)
path	No	Path on the server	/
domain	No	Domain name	Current domain
secure	No	Send over HTTPS only	false
httponly	No	Prevent JS access	false

Note: setcookie() must be called before any output (HTML).

2. CREATE COOKIE EXAMPLES

Simple Cookie (Session Cookie)

```php
<?php
setcookie("theme", "dark"); // Session cookie
?>
```

Cookie with Expiry (1 Day)

```php
<?php
setcookie("username", "Lovejeet", time() + 86400, "/");
// 86400 = 24 hours
?>
```

Cookie with All Parameters

```php
<?php
setcookie("user", "Lovejeet", time() + 604800, "/",
    ".exemple.com", true, true);
 // 604800 = 7 days
?>
```

3. ACCESS COOKIE USING $_COOKIE

Use $_COOKIE associative array to read cookie values.

```php
<?php
if(isset($_COOKIE['username'])){
   echo "Welcome, " . $_COOKIE['username'];
} else {
   echo "Cookie not set!";
}
?>
```

4. CHECK IF COOKIE EXISTS

```php
<?php
if(isset($_COOKIE['theme'])){
   echo "Theme: " . $_COOKIE['theme'];
}
?>
```

5. COOKIE PROPERTIES EXPLAINED

Property	Explanation
Cookie Name	Unique name (e.g., username, theme)
Value	Data stored (e.g., Lovejeet, dark)
Expiry	Timestamp when cookie expires. 0 = session cookie (deleted on browser close).
Path	URL path where cookie is available. Default is "/".
Secure	If true, cookie sent only via HTTPS. Default is false.
HttpOnly	If true, cookie cannot be accessed via JavaScript (more secure).

6. EXAMPLE: LOGIN STATE WITH COOKIE

Store login state for 7 days.

```php
<?php
// Login
setcookie("login", "true", time() + 604800, "/");

// Check login
if(isset($_COOKIE['login']) && $_COOKIE['login'] == "true"){
   echo "User is logged in ✅";
} else {
   echo "Please login first ❌";
}
?>
```

7. DELETE COOKIE

Set expiry time in the past.

```php
<?php
setcookie("username", "", time() - 3600, "/");
// Deletes the cookie
?>
```

Note: Deleting removes cookie from the browser.

8. COMMON COOKIE USE CASES

Remember Username — Keep user logged in.

User Preferences — Store theme, language, layout settings.

Shopping Cart — Store cart items for guest users.

Analytics & Tracking — Track visits, pages, and user behavior.

9. BEST PRACTICES

- Always use meaningful cookie names.
- Set expiry time wisely.
- Use secure cookies on HTTPS.
- Use HttpOnly to prevent XSS.
- Do not store sensitive data in cookies (use sessions instead).

QUICK REFERENCE

Task	Code
Create Cookie	`setcookie("name","value");`
Read Cookie	`$_COOKIE["name"];`
Check Cookie	`isset($_COOKIE["name"]);`
Delete Cookie	`setcookie("name","",time()-3600,"/");`
Session Cookie	`Expiry = 0 (default)`
1 Day Expiry	`time() + 86400`
Secure Cookie	`secure = true`
HttpOnly Cookie	`httponly = true`

IMPORTANT TIP: Cookies are stored on the client side. Anyone can view and modify them. Never store sensitive information like passwords in cookies.

PHP JSON

Work with JSON Data Easily!

```php
<?php
$json = json_encode($data);
$data = json_decode($json, true);
?>
```

json_encode() — Converts PHP data into JSON string.

json_decode() — Converts JSON string into PHP data.

JSON Object — Data in key-value pairs (like object).

JSON Array — Data in ordered list (like array).

API Data — JSON is commonly used to send/receive data.

Encode / Decode — Encode PHP → JSON, Decode JSON → PHP

Response — JSON is used in API responses.

1. JSON OBJECT (Example)

JSON Object contains key-value pairs.

```
{
  "name": "Lovejeet",
  "age": 32,
  "city": "Amritsar",
  "isAdmin": true
}
```

2. JSON ARRAY (Example)

JSON Array contains ordered list of values.

```
[
  "PHP",
  "MySQL",
  "JavaScript",
  "HTML",
  "CSS"
]
```

3. ENCODE (PHP → JSON)

Convert PHP data into JSON string.

PHP Array / Object:

```
$data = [
  'name' => 'Lovejeet',
  'age'  => 32,
  'city' => 'Amritsar'
];
```

Encode:

```
$json = json_encode($data, JSON_PRETTY_PRINT);
```

Result (JSON):

```
{
  "name": "Lovejeet",
  "age": 32,
  "city": "Amritsar"
}
```

4. DECODE (JSON → PHP)

Convert JSON string into PHP data.

JSON String:

```
{
  "name": "Lovejeet",
  "age": 32,
  "city": "Amritsar"
}
```

Decode to Associative Array:

```
$data = json_decode($json, true);
```

Decode to Object:

```
$data = json_decode($json, false);
```

Result (PHP Array):

```
Array
(
    [name] => Lovejeet
    [age] => 32
    [city] => Amritsar
)
```

5. API RESPONSE (Example)

Common JSON response from API.

```
{
  "status": "success",
  "code": 200,
  "message": "Data fetched successfully",
  "data": [
    {
      "id": 1,
      "name": "Lovejeet",
      "email": "lovejeet@example.com"
    },
    {
      "id": 2,
      "name": "Jeet",
      "email": "jeet@example.com"
    }
  ]
}
```

6. COMMON JSON FUNCTIONS

Function	Purpose
json_encode($data)	Convert PHP data to JSON string
json_decode($json, true)	Convert JSON to PHP Associative Array
json_decode($json, false)	Convert JSON to PHP Object
json_last_error()	Get last JSON error
json_last_error_msg()	Get last JSON error message
JSON_PRETTY_PRINT	Make JSON output readable (formatted)
JSON_UNESCAPED_UNICODE	Keep Unicode characters as it is

7. WORKFLOW (API REQUEST & RESPONSE)

Data Format: JSON

Fast • Lightweight • Easy to Read • Platform Independent

8. FETCH API DATA IN PHP (Example)

Get JSON data from an API and display.

```php
<?php
$url = 'https://jsonplaceholder.typicode.com/users/1';
$response = file_get_contents($url);
$data = json_decode($response, true);

echo "<h3>Name: " . $data['name'] . "</h3>";
echo "Email: " . $data['email'];
?>
```

Sample API Response (JSON):

```
{
  "id": 1,
  "name": "Leanne Graham",
  "email": "leanne@example.com",
  "phone": "1-770-736-8031",
  "website": "hildegard.org"
}
```

9. BEST PRACTICES

- Always use json_last_error() to check errors.
- Use JSON_PRETTY_PRINT for readable output.
- Use true in json_decode() for associative array.
- Always validate JSON data from external APIs.
- Use HTTPS for secure API communication.

10. IMPORTANT NOTE

- JSON is language independent.
- Lightweight and fast for data exchange.
- Ideal for APIs, web services and AJAX.
- Always handle errors while encoding/decoding.

QUICK REFERENCE

Task	Code
Encode PHP → JSON	json_encode($data)
Decode JSON → Array	json_decode($json, true)
Decode JSON → Object	json_decode($json, false)
Check JSON Error	json_last_error()
Get Error Message	json_last_error_msg()
Pretty JSON Output	JSON_PRETTY_PRINT
Unicode Support	JSON_UNIESCAPED_UNICODE

TIP: JSON is widely used in REST APIs, AJAX requests, and real-time web applications. Learn once, use everywhere!

PHP REGULAR EXPRESSIONS

Powerful Pattern Matching, Search, Replace & Validation!

```php
<?php
preg_match('/pattern/', $subject, $matches);
$new = preg_replace('/pattern/', 'replacement', $subject);
$parts = preg_split('/pattern/', $subject);
?>
```

preg_match()
Search a pattern in a string and return match result.

preg_replace()
Search a pattern and replace with another string.

preg_split()
Split a string into an array using a pattern as delimiter.

Pattern
The rules (syntax) used to search in a string.

Regex
Short for Regular Expression. A powerful pattern.

Match
When the pattern is found in the string.

Replace
Replace matched text with another string using regex.

Validation
Check if a string is valid using regex patterns.

1. preg_match() – Find a Match

Search a pattern in a string.

```php
<?php
$pattern = '/\d+/';   // match one or more digits
$text    = 'Order no: 12345';
if (preg_match($pattern, $text, $matches)) {
   echo 'Matched: ' . $matches[0]; // 12345
} else {
   echo 'No match found.';
}
?>
```

$matches[0] = Full match
$matches[1], $matches[2], ... = Captured groups

2. preg_replace() – Replace Text

Replace matched pattern with another string.

```php
<?php
$pattern = '/\s+/';       // match whitespace
$text    = 'Hello   World   PHP';
$replace = ' ';
$newText = preg_replace($pattern, $replace, $text);
echo $newText; // Hello World PHP
?>
```

Use it to clean, format or modify strings.

3. preg_split() – Split String

Split string into array using a pattern.

```php
<?php
$pattern = '/[,;\s]+/'; // comma, semicolon or space
$text    = 'apple, banana; mango   orange';
$parts   = preg_split($pattern, $text);
print_r($parts);
?>
```

Output:

```
Array
(
    [0] => apple
    [1] => banana
    [2] => mango
    [3] => orange
)
```

4. Common Patterns (Regex Cheat Sheet)

Pattern	Meaning	Example
.	Any single character	a.c matches abc, a1c
\d	Digit (0-9)	\d\d matches 12
\D	Non-digit	\D matches a, $
\w	Word character (a-z, A-Z, 0-9, _)	\w+ matches PHP123
\W	Non-word character	\W matches @, #
\s	Whitespace (space, tab, \n)	a\sb matches a b
\S	Non-whitespace	a\Sb matches a,b
^	Start of string	^Hello matches Hello World
$	End of string	World$ matches Hello World
[]	Character set	[abc] matches a or b or c
[^]	Negated set	[^0-9] matches non-digits
()	Capturing group	(abc) captures abc
|	OR operater	cat|dog matches cat or dog
*	0 or more	ab*c matches ac, abc, abbc
+	1 or more	ab+c matches abc, abbc
?	0 or 1	ab?c matches ac or abc
{n}	Exactly n times	a{3} matches aaa
{n,}	At least n times	a{2,} matches aa, aaa
{n,m}	Between n and m	a{2,4} matches aa, aaa, aaaa

5. Validation Examples

Validate common input using regex.

Validation Type	Pattern	Example (Valid)
Email	/^[^\s@]+@[^\s@]+\.[^\s@]+$/	user@mail.com
Mobile (10 digit)	/^[6-9]\d{9}$/	9876543210
URL	/^https?:\/\/[^\s\/$.?#].[^\s]*$/	https://example.com
Date (DD/MM/YYYY)	/^(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/(19|20)\d{2}$/	25/12/2024
IP Address	/^(\d{1,3}\.){3}\d{1,3}$/	192.168.1.1

6. Real-life Usage Examples

- Form validation (email, phone, password)
- Search and extract data (order ID, invoice no.)
- Clean unwanted characters from text
- Create slugs (SEO friendly URLs)
- Split and process CSV/log data

7. Full Example: Validate Email

Check if email is valid using preg_match().

```php
<?php
$email   = 'test@example.com';
$pattern = '/^[^\s@]+@[^\s@]+\.[^\s@]+$/';
if (preg_match($pattern, $email)) {
   echo 'Valid Email ✅';
} else {
   echo 'Invalid Email ❌';
}
?>
```

preg_match() returns 1 (true) if matched, 0 (false) if not matched.

8. Replace HTML Tags (Example)

Remove all HTML tags from a string.

```php
<?php
$text  = '<h1>Hello <b>World</b></h1><p>PHP is <i>great</i>!</p>';
$clean = preg_replace('/<[^>]*>/', '', $text);
echo $clean; // Hello WorldPHP is great!
?>
```

/<[^>]*>/ matches anything between < and > (HTML tags).

9. Split by Multiple Delimiters

Split string by comma, semicolon or pipe.

```php
<?php
$text  = 'one, two; three|four five';
$parts = preg_split('/[,;|\s]+/', $text);
print_r($parts);
?>
```

Result:

```
Array
  [0] => one  [1] => three  [2] => four  [4] => five
)
```

10. Best Practices

- Always test your pattern before using.
- Use delimiters (/pattern/) for all regex.
- Escape special characters if needed (e.g., \. for dot).
- Keep patterns simple and readable.
- Validate and sanitize user input.

11. important Note

- Regex is case sensitive by default.
- Use modifiers like i (case-insensitive), m (multi-line), s (dotall).
- Test your regex: https://regex101.com

Pro Tip: Start simple, test often, and build complex patterns step by step. Practice makes perfect! 🚀

PHP ERROR HANDLING

Detect, Handle & Debug Errors Like a Pro!

```php
<?php
error_reporting(E_ALL);
ini_set('display_errors', 1);
trigger_error('This is a custom error!', E_USER_WARNING);
?>
```

error_reporting() — Set which error levels are reported.

trigger_error() — Generate user-defined errors or warnings.

Error — Serious issues that break the script.

Warning — Runtime issues that don't stop the script.

Notice — Informational messages about the script.

Fatal Error — Critical errors that stop script execution.

Error Level — Numeric values that define error severity.

Debugging — Find, fix and improve your code.

1. error_reporting() – Set Error Levels

Control which PHP errors are reported.

```php
<?php
error_reporting(E_ALL);                 // Report all errors
error_reporting(E_ERROR | E_WARNING);   // Only Error & Warning
error_reporting(0);                     // Hide all errors
?>
```

Tip: Use E_ALL during development, and lower levels in production.

2. Common Error Levels

Constant	Value	Description	Example
E_ERROR	1	Fatal run-time errors.	Undefined function
E_WARNING	2	Run-time warnings.	Division by zero
E_PARSE	4	Compile-time parse errors.	Syntax error
E_NOTICE	8	Run-time notices.	Undefined variable
E_STRICT	16	Strict standards notices.	Deprecated usage
E_DEPRECATED	8192	Deprecated code warnings.	Old function usage
E_USER_ERROR	256	User-generated error.	trigger_error(..., E_USER_ERROR)
E_USER_WARNING	512	User-generated warning.	trigger_error(..., E_USER_WARNING)
E_USER_NOTICE	1024	User-generated notice.	trigger_error(..., E_USER_NOTICE)
E_ALL	32767	All of the above.	All errors & warnings

3. trigger_error() – Generate Custom Errors

Create custom error messages.

```php
<?php
trigger_error("This is a custom error!", E_USER_ERROR); // Error
trigger_error("This is a warning!", E_USER_WARNING);  // Warning
trigger_error("This is a notice.", E_USER_NOTICE);    // Notice
?>
```

User Error Types

- E_USER_ERROR → Stops script execution
- E_USER_WARNING → Script continues
- E_USER_NOTICE → Informational message

4. Display Errors (Development vs Production)

Show or hide errors on screen.

Development (Show Errors)

```php
<?php
ini_set('display_errors', 1);
error_reporting(E_ALL);
?>
```

Production (Hide Errors)

```php
<?php
ini_set('display_errors', 0);
error_reporting(0);
?>
```

Best Practice:
Enable error display only in development environment.

5. Example: Different Error Types

Examples of different PHP errors.

```php
<?php
echo $a;                // Notice: Undefined variable $a
echo 10/0;              // Warning: Division by zero
include 'file.php';     // Warning: Failed to open stream
function test(){
   return;
}
test();
echo undefined_function();  // Fatal Error: Call to undefined function
?>
```

6. Error Reporting Level Chart

Level	Description	Use Case
0	No errors reported	Production (hide all)
E_ERROR	Only fatal errors	Minimal reporting
E_ERROR | E_WARNING	Errors & warnings	Basic debugging
E_ALL	All errors, warnings, notices	Full debugging (development)

7. Custom Error Handler (Advanced)

Handle errors using a custom function.

```php
<?php
function customErrorHandler($errno, $errstr, $errfile, $errline) {
    echo "<b>Error [$errno]</b>: $errstr in $errfile on line $errline";
    echo "<br>";
}
set_error_handler('customErrorHandler');
?>
```

set_error_handler() allows you to define your own error handler.

8. Common Causes of Errors

- Undefined variables or functions
- Syntax mistakes
- Wrong file paths or permissions
- Division by zero
- Deprecated functions
- Database connection issues

9. Debugging Tips

- Enable error reporting during development.
- Read error messages carefully.
- Use var_dump(), print_r() for debugging.
- Check logs (error_log) for hidden errors.
- Test code in small parts.

10. Useful Functions for Debugging

Function	Purpose
error_get_last()	Get last occurred error
error_clear_last()	Clear last error
ini_set()	Set configuration options
debug_backtrace()	Get stack trace of errors
var_dump(), print_r()	Inspect variables

11. Important Note

Never show errors in production environment. It may expose sensitive information about your application.

Pro Tip: Good error handling makes your code reliable and easier to debug. Always fix the root cause, not just the error!

PHP EXCEPTIONS

Handle Errors Gracefully & Write Better Code!

```php
<?php
try {
    // risky code that may throw exception
} catch (Exception $e) {
    // handle exception
}
```

try — Wrap risky code that may fail.

catch — Catch & handle the exception.

finally — Always runs after try & catch block.

throw — Throw a custom exception.

Exception — Base class for all exceptions.

Throwable — Root interface for Exception & Error.

Custom Exception — Create your own exception class.

Error Handling — Write clean, safe & reliable code.

1. Basic Exception Handling

Use try, catch & finally to handle exceptions.

```php
<?php
try {
    $result = 10 / 0;
    echo $result;
} catch (Exception $e) {
    echo "Caught Exception: " . $e->getMessage();
} finally {
    echo "This will always run.";
}
?>
```

Tip: finally block is optional but useful for cleanup (files, DB, connections).

2. Throwing an Exception

Use throw to raise an exception.

```php
<?php
function divide($a, $b) {
    if ($b == 0) {
        throw new Exception("Division by zero!");
    }
    return $a / $b;
}

try {
    echo divide(10, 0);
} catch (Exception $e) {
    echo $e->getMessage();
}
?>
```

throw stops the current flow and jumps to catch block.

3. Multiple Catch Blocks

You can catch different types of exceptions.

```php
<?php
try {
    $file = fopen("unknown.txt", "r");
    if (!$file) {
        throw new Exception("File not found");
    }
} catch (Error $err) {
    echo "Error: " . $err->getMessage();
} catch (Exception $e) {
    echo "Exception: " . $e->getMessage();
}
?>
```

Order matters: catch specific exceptions first.

4. finally Block (Always Executes)

Used for cleanup code (files, DB, resources).

```php
<?php
$fp = fopen("test.txt", "r");
try {
    if (!$fp) {
        throw new Exception("Unable to open file");
    }
    echo "File opened successfully.";
} catch (Exception $e) {
    echo $e->getMessage();
} finally {
    if ($fp) fclose($fp);  // always close file
    echo "\nFile closed.";
}
?>
```

Runs even if exception occurs or not.

5. Built-in Exception & Throwable

Exception and Error both implement Throwable.

Type	Description	Example
Exception	Base class for user errors	Exception, RuntimeException
Error	Serious system errors	ParseError, TypeError
Throwable	Root interface	Exception, Error

Use catch (Throwable $t) to catch both Exception & Error.

6. Custom Exception

Create your own exception class.

```php
<?php
class AgeException extends Exception {
    public function __construct($message) {
        parent::__construct($message);
    }
}

function checkAge($age) {
    if ($age < 18) {
        throw new AgeException("You must be 18+ to continue.");
    }
    echo "Access granted!";
}
?>
```

Extending Exception helps to create meaningful error types.

7. Common Exception Methods

Method	Description
getMessage()	Returns exception message
getCode()	Returns exception code
getFile()	Returns file where exception occurred
getLine()	Returns line number
getTrace()	Returns stack trace
getTraceAsString()	Returns stack trace as string
getPrevious()	Returns previous exception

Use these methods to debug and log exceptions.

8. Common Exceptions in PHP

Exception	When it Occurs
Exception	General runtime errors
InvalidArgumentException	Invalid function argument
RuntimeException	Runtime related errors
OutOfRangeException	Value out of allowed range
LogicException	Bad logic in the code
DomainException	Domain/Business rule violation

Choose the right exception for the right situation.

9. Best Practices

- Use exceptions for exceptional conditions, not for control flow.
- Catch specific exceptions first, then general ones.
- Always clean up resources in finally block.
- Create custom exceptions for your application.
- Log exceptions for debugging and monitoring.

10. Example: Complete Flow

```php
<?php
try {
    $user = ['name' => 'Alice', 'age' => 16];
    if ($user['age'] < 18) {
        throw new AgeException("Age must be 18 or above.");
    }
    echo "Welcome " . $user['name'];
} catch (AgeException $e) {
    echo "Error: " . $e->getMessage();
} catch (Throwable $t) {
    echo "Something went wrong: " . $t->getMessage();
} finally {
    echo "\nThank you for visiting.";
}
?>
```

Output:

```
Error: Age must be 18 or above.
Thank you for visiting.
```

11. Important Note

Never suppress exceptions silently. Always log or display meaningful messages (not in production).

Good exception handling = Better user experience + Easier debugging!

12. Pro Tip

Use exceptions to write clean, safe and maintainable code. A well-handled error is a better application!

PHP OOP BASICS

Object Oriented Programming in PHP Made Simple!

```php
<?php
// OOP makes code reusable, organized & easy to maintain
// Think in terms of real-world objects!
?>
```

Class — Blueprint for creating objects.

Object — Instance of a class.

Property — Variables inside a class.

Method — Functions inside a class.

$this — Refers to the current object.

new — Keyword to create an object.

public — Accessible from anywhere.

private — Accessible only inside the class.

1. Class

A class is a blueprint or template for creating objects.

```php
<?php
class Car {
    // Class body

?>

```

Use class keyword to declare a class.
Class name should start with a capital letter.

2. Object

An object is an instance of a class.

```php
<?php
class Car {
    public $brand = "Honda";
}

$car = new Car();   // Object of Car class
?>
```

Use new keyword to create an object.
$car is an object (instance) of Car class.

3. Property

Properties are variables that belong to a class.

```php
<?php
class Car {
    public $brand = "Honda";
    private $model = "Civic";
}
?>
```

Properties store data about the object.
Access using $object->propertyName

4. Method

Methods are functions that belong to a class.

```php
<?php
class Car {
    public function start() {
        echo "Car started!";
    }
}
?>
```

Methods define behavior of the object.
Call using $object->methodName()

5. $this

$this refers to the current object inside a method.

```php
<?php
class Car {
    public $brand = "Honda";

    public function showBrand() {
        echo $this->brand; // refers to current object
    }
}
?>
```

$this helps access properties and methods of the current object.

6. new

new keyword is used to create an object.

```php
<?php
class Car {
    public $brand = "Honda";
}

$car = new Car();   // Create object
$car2 = new Car();  // Another object
?>
```

Each new object has its own copy of properties.

7. public

Public members are accessible from anywhere.

```php
<?php
class Car {
    public $brand = "Honda";
    public function start() {
        echo "Starting...";
    }
}
$car = new Car();
echo $car->brand;       // OK
$car->start();          // OK
?>
```

Use public to allow full access.

8. private

Private members are accessible only inside the class.

```php
<?php
class Car {
    private $model = "Civic";
    private function secret() {
        echo "This is private!";
    }
}
$car = new Car();
// echo $car->model;   // Error
// $car->secret();     // Error
?>
```

Use private to hide data and protect your code.

9. Example: Complete Code

```php
<?php
class Person {
    public $name;
    private $age;

    public function __construct($name, $age) {
        $this->name = $name;
        $this->age = $age;
    }
    public function introduce() {
        echo "Hi, I am {$this->name}";
    }
}
$p = new Person("Amit", 21);
$p->introduce();
?>
```

Output:

```
Hi, I am Anit
```

Constructors (__construct) run automatically when object is created.

10. Summary Table

Concept	Keyword / Symbol	Example	Purpose
Class	class	class Car {}	Blueprint for objects
Object	new	$car = new Car();	Create an object
Property	$property	$brand = "Honda";	Store data
Method	function	function start() {}	Define behavior
$this	$this	$this->brand	Refers current object
public	public	public $name;	Accessible everywhere
private	private	private $name;	Accessible inside class only

11. Benefits of OOP

- **Code Reusability** — Write once, use many times.
- **Better Organization** — Keep code clean and structured.
- **Easy Maintenance** — Modify one part without breaking others.
- **Scalability** — Handle big projects easily.

Pro Tip: Always think in terms of real-world objects. That's the power of OOP!

Note: OOP is optional in PHP, but using it makes your code more powerful & professional.

Best Practice: Use OOP when building medium to large applications for better structure and reusability.

PHP CONSTRUCTORS & DESTRUCTORS

Create, Initialize & Clean Objects the Right Way!

```php
<?php
// Constructors run when object is created
// Destructors run when object is destroyed
?>
```

__construct() — Special method that runs automatically on object creation.

__destruct() — Special method that runs automatically on object destruction.

Object Creation — Creates instance of a class using *new* keyword.

Initialization — Setup object properties and perform initial tasks.

Cleanup — Release resources and perform final tasks.

Parameters — Pass values to constructor to customize object creation.

Properties — Store data inside objects using properties.

Methods — Functions inside class that define object behavior.

1. What is Constructor?

Constructor is a special method that is called automatically when an object is created.

```php
<?php
class User {
    public $name;
    public $email;

    // Constructor
    public function __construct($name, $email) {
        $this->name  = $name;
        $this->email = $email;
    }
}
$user = new User("Amit", "amit@example.com");
?>
```

Runs automatically when *new User()* is called.

2. What is Destructor?

Destructor is a special method that is called automatically when an object is destroyed or script ends.

```php
<?php
class Logger {
    public function __construct() {
        echo "Logger started\n";
    }
    public function __destruct() {
        echo "Logger closed\n";
    }
}
$log = new Logger();
// Destructor will run automatically
?>
```

Runs automatically when object is destroyed.

3. Object Creation & Initialization

Use *new* keyword to create object and initialize using constructor.

```php
<?php
class Product {
    public $name, $price;
    public function __construct($name, $price) {
        $this->name  = $name;
        $this->price = $price;
    }
}
$p = new Product("Laptop", 45000);
echo $p->name . " - " . $p->price;
?>
```

Object created and properties initialized in constructor.

4. Parameterized Constructor

You can pass parameters to constructor to set initial values.

```php
<?php
class Student {
    public $id, $name, $course;
    public function __construct($id, $name, $course) {
    {  $this->id     = $id;
       $this->name   = $name;
       $this->course = $course;
    }
}
$s = new Student(101, "Rohit", "PHP");
?>
```

Helps in creating fully initialized objects.

5. Properties & Methods in Action

Use properties to store data and methods to process it.

```php
<?php
class Account {
    public $owner;
    private $balance = 0;

    public function __construct($owner, $balance) {
        $this->owner   = $owner;
        $this->balance = $balance;
    }
    public function deposit($amount) {
        $this->balance += $amount;
    }
    public function getBalance() {
        return $this->balance;
    }
    public function __destruct() {
        echo "Account of {$this->owner} closed.\n";
    }
}
$acc = new Account("Anita", 1000);
$acc->deposit(500);
echo "Balance: " . $acc->getBalance();
?>
```

Output:

```
Balance: 1500
Account of Anita closed.
```

Key Points

- Properties hold data.
- Methods define behavior.
- Constructor initializes.
- Destructor cleans up.

6. Constructor vs Destructor

Feature	Constructor (__construct)	Destructor (__destruct)
Purpose	Initialize object	Cleanup before object is destroyed
When it runs	When object is created	When object is destroyed or script ends
How it's called	Automatically	Automatically
Can have parameters?	Yes	No
Return type	No return	No return
Visibility	Usually public	Usually public

7. Common Use Cases

- Database connection open in constructor and close in destructor.
- File handle open in constructor and close in destructor.
- Initialize configuration, settings, or default values.
- Free memory, close sessions, release resources.

8. Quick Reference

Term	Description
__construct()	Called automatically when object is created.
__destruct()	Called automatically when object is destroyed.
new	Creates a new object (instance) of a class.
Initialization	Setting initial values using constructor.
Cleanup	Releasing resources using destructor.

Pro Tip: Always initialize properties in constructor and release resources in destructor.

Note: Destructor is not guaranteed in case of sudden script termination or fatal error.

Best Practice: Keep constructor light. Use destructor for proper cleanup only.

Important: Destructor has no parameters and cannot be called manually.

PHP INHERITANCE

Extend. Reuse. Override. Build Better!

```php
<?php
// Inheritance lets you create a new class
// based on an existing class using extends
?>
```

extends — Keyword used to inherit a class.

Parent Class — Base class whose properties & methods are inherited.

Child Class — Derived class that inherits from parent class.

Method — Functions defined inside class.

Property — Variables defined inside class.

Override — Redefine parent method in child class.

Reuse Code — Inherit & reuse code, reduce redundancy.

Inheritance — One class acquires properties & behavior of another.

1. What is Inheritance?

Inheritance allows a class (child class) to inherit the properties and methods of another class (parent class).

```php
<?php
class Animal {
    public $type = "Animal";
    public function eat() {
        echo "Animal is eating.";
    }
}
```

Animal is the parent class.

2. Basic Syntax

Use extends keyword to inherit a class.

```php
<?php
class Dog extends Animal {
    public function bark() {
        echo "Dog is barking.";
    }
}
?>
```

Dog is the child class that inherits from Animal.

3. Access Inherited Members

Child class can access parent's properties & methods.

```php
<?php
$dog = new Dog();
echo $dog->type;   // Animal
$dog->eat();       // Animal is eating.
$dog->bark();      // Dog is barking.
?>
```

Child class can use inherited members directly.

4. Method Override

Redefine parent method in child class.

```php
<?php
class Dog extends Animal {
    public function eat() {
        echo "Dog is eating meat.";
    }
}
?>
```

Child class method overrides parent method.

5. Access Parent Method (parent::)

Use parent:: to call parent class method.

```php
<?php
class Dog extends Animal {
    public function eat() {
        parent::eat();
        echo "  Dog prefers meat.";
    }
}
?>
```

Call parent class method using parent::method().

6. Parent vs Child

Parent Class	Child Class
Base class	Derived class
Defines common properties & methods	Inherits and can add new features
Does not depend on child	Depends on parent
General concept	Specific implementation

Child class is a specialized version of parent class.

7. Example: Complete Code

```php
<?php
class Vehicle {
    public $wheels = 4;
    public function move() {
        echo "Vehicle is moving.";
    }
}
class Car extends Vehicle {
    public $brand;
    public function __construct($brand) {
        $this->brand = $brand;
    }
    public function move() {
        parent::move();
        echo " Car brand: " . $this->brand . "\n";
    }
}
$car = new Car("Toyota");
echo $car->wheels . " wheels\n";  // 4 wheels
$car->move();
?>        // Vehicle is moving. Car brand: Toyota
```

Output:

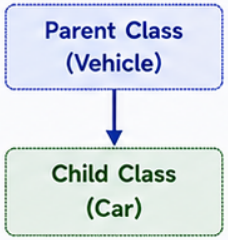
```
4 wheels
Vehicle is moving.
Car brand: Toyota
```

Key Points

- Car inherits wheels and move() from Vehicle.
- move() is overridden.
- parent::move() calls parent method.

8. Inheritance Types in PHP

PHP supports single inheritance.



- PHP does not support multiple inheritance directly.
- It can be achieved using interfaces.

9. Benefits of Inheritance

- **Code Reusability** – Reuse existing code, reduce duplication.
- **Better Organization** – Create a clear hierarchy.
- **Easier Maintenance** – Change in parent reflects in child.
- **Extensibility** – Add new features in child class.
- **Consistency** – Maintain common behavior.

10. Quick Reference

Term	Description
extends	Keyword used to inherit a class.
Parent Class	The base class being inherited.
Child Class	The class that inherits from parent.
Method Override	Redefine parent class method in child class.
parent::	Used to access parent class members.
Inheritance	Mechanism to acquire properties & methods from another class.

Pro Tip: Always design parent class with common behavior and child class with specific behavior.

Note: Constructor is not inherited automatically. Define it in child class if needed.

Best Practice: Use inheritance only when there is a clear "is-a" relationship.

Important: Avoid deep inheritance chains. Keep it simple and clean.

PHP ENCAPSULATION

Hide Data. Control Access. Write Better Code!

```php
<?php
// Encapsulation = Data Hiding + Access Control
// Protect your data. Control how it's accessed.
// Build secure & maintainable applications.
?>
```

public — Accessible from anywhere.

private — Accessible only inside the class.

protected — Accessible inside the class and its children.

Getters — Methods used to read private data.

Setters — Methods used to modify private data.

Access Control — Restrict direct access to members.

Data Hiding — Hide internal details & show only what's needed.

Encapsulation — Binding data & methods together in a class.

1. What is Encapsulation?

Encapsulation is the concept of wrapping data (properties) and methods together and controlling access to the data.

```php
<?php
class BankAccount {
    private $balance;       // hidden data
    public function getBalance() {
        return $this->balance;
    }
}
?>
```

Data is hidden and accessed through methods.

2. Access Modifiers in PHP

PHP provides 3 types of access modifiers.

Modifier	Scope	Access From
public	No restrictions	Anywhere
private	Only inside class	Only inside class
protected	Inside class and subclasses	Class & Child classes

By default, class members are public.

3. Example: Access Modifiers

Example class using public, private & protected.

```php
<?php
class User {
    public $name;          // public property
    private $email;        // private property
    protected $role;       // protected property
}
?>
```

Access based on modifier rules.

4. Why Use Encapsulation?

- Protect data from direct access.
- Prevent accidental modification.
- Improve code security.
- Better control over data.
- Easier to manage and maintain code.

Encapsulation = Secure + Maintainable Code.

5. Getters (Accessors)

Use getters to read private properties.

```php
<?php
class Product {
    private $price;
    public function getPrice() {
        return $this->price;
    }
}
?>
```

Getters provide read-only access to data.

6. Setters (Mutators)

Use setters to modify private properties.

```php
<?php
class Product {
    private $price;
    public function setPrice($p) {
        if ($p > 0) {
            $this->price = $p;
        }
    }
}
?>
```

Setters control how data can be modified.

7. Complete Example

Encapsulation with access control.

```php
<?php
class Employee {
    private $name;          // hidden property
    private $salary;

    public function setName($n) { $this->name = $n; }
    public function getName() { return $this->name; }

    public function setSalary($s) {
        if ($s > 0) { $this->salary = $s; }
    public function getSalary() { return $this->salary; }
}
?>
```

Usage:

```php
$e = new Employee();
$e->setName("Anita");
$e->setSalary(50000);
echo $e->getName();  // Anita
echo $e->getSalary(); // 50000
```

Properties accessed only through methods.

8. Access Control Summary

Where can each modifier be accessed?

Modifier	Inside Class	Child Class	Outside Class
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No

Choose the right modifier for the right reason.

9. Real Life Analogy

Think of a Bank Account:

- Balance (private) → Hidden from users
- Deposit (setter) → Add money
- Withdraw (setter) → Remove money
- Check Balance (getter) → Show balance
- You control how data is accessed.

10. Best Practices

- Use private for properties by default.
- Use getters & setters for data access.
- Validate data inside setters.
- Never expose sensitive data directly.
- Follow the principle of least privilege.

11. Quick Reference

Term	Description
public	Accessible from anywhere.
private	Accessible only inside the class.
protected	Accessible in class and child classes.
Getter	Method to read private data.
Setter	Method to modify private data.
Encapsulation	Binding data and methods with access control.

12. Important Note

Encapsulation is not about hiding code, it is about protecting data and controlling access.

Pro Tip: Always keep data private and provide public methods (getters/setters) to access it safely.

Note: Protected is useful for inheritance. Private is best for strict data hiding.

Best Practice: Encapsulation makes your code secure, clean, flexible and easy to maintain.

PHP INTERFACES

Define a Contract. Implement. Build Strong & Scalable Code!

```php
<?php
// Interfaces define a contract (methods only)
// Classes implement the contract using implements
// Achieve abstraction & multiple inheritance
?>
```

interface — Define a blueprint of methods.

implements — Class uses interface methods.

Methods — Interfaces declare method signatures.

Contract — Interface is a contract that must be followed.

Multiple Interfaces — A class can implement multiple interfaces.

Abstraction — Focus on what, not how.

Class — Class implements and provides structure.

Structure — Interfaces help organize code structure.

1. What is Interface?

An interface is a contract that contains only method declarations. Classes implement interfaces and provide the method definitions.

```php
<?php
interface Shape {
    public function area();
    public function perimeter();
}
?>
```

Interfaces cannot have properties or method bodies.

2. Basic Syntax

Use interface to declare and implements to use.

```php
<?php
interface Animal {
    public function sound();
}

class Dog implements Animal {
    public function sound() {
        echo "Bark";
    }
}
?>
```

A class must define all methods of the interface.

3. Multiple Interfaces

A class can implement more than one interface.

```php
<?php
interface Flyable {
    public function fly();
}
interface Swimmable {
    public function swim();
}

class Duck implements Flyable, Swimmable {
    public function fly() { echo "Flying"; }
    public function swim() { echo "Swimming"; }
}
?>
```

Separate interfaces by comma.

4. Interface vs Class

Key differences between interface and class.

Interface	Class
Contains only method declarations ✓	Can have properties & methods ✗
Methods are public (by default) ✓	Methods can be public, private, etc. ✗
Cannot have properties ✓	Can have properties ✗
Achieves abstraction ✓	Provides structure & implementation ✓
Multiple inheritance is possible ✓	Does not support multiple inheritance ✗

5. Example: Simple Interface

Example of interface and its implementation.

```php
<?php
interface Car {
    public function start();
    public function stop();
}

class Toyota implements Car {
    public function start() {
        echo "Toyota started";
    }
    public function stop() {
        echo "Toyota stopped";
    }
}
?>
```

Toyota class implements all methods of Car interface.

6. Rules of Interfaces

- Method names must match exactly.
- Method visibility must be public.
- A class must implement all interface methods.
- Interfaces can extend other interfaces.
- Interfaces cannot be instantiated.

7. Interface Extends Interface

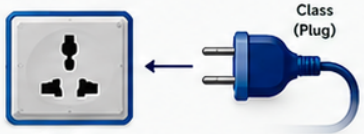
One interface can extend another interface.

```php
<?php
interface Animal {
    public function eat();
}

interface Pet extends Animal {
    public function play();
}

class Cat implements Pet {
    public function eat() { echo "Eating"; }
    public function play() { echo "Playing"; }
}
?>
```

Pet interface extends Animal interface.

8. Real Life Analogy

Interface is like a plug point.



Different plugs (classes) can be used in the same socket (interface) as long as they follow the same standard.

9. Benefits of Interfaces

- Promotes loose coupling
- Enforces a standard contract
- Supports multiple inheritance
- Improves code organization
- Makes code more flexible and maintainable

10. Quick Reference

Term	Description
Interface	Defines a contract of methods.
implements	Class uses interface methods.
Methods	Method signatures without body.
Contract	Rules that must be followed.
Multiple Interfaces	Class can implement many interfaces.
Abstraction	Hide details, show functionality.
Class	Provides structure and behavior.
Structure	Organized and reusable code.

Pro Tip: Use interfaces when you want to define what a class can do, not how it does.

Note: Interface methods are public by default. PHP 8.1+ allows constants & default methods in interfaces.

Best Practice: Program to an interface, not to an implementation.

Important: A class must implement all methods of an interface, otherwise it will throw Fatal Error.

PHP TRAITS

Write Less. Reuse More. Share Code Smartly!

```php
<?php
// Traits allow code reusability in single inheritance
// Use traits to include methods in classes
// Promote code sharing & clean structure
?>
```

trait — Define a set of methods.

use — Use trait inside a class.

Reusable Methods — Write once, use many times.

Code Sharing — Share methods across classes.

Multiple Traits — Use multiple traits in one class.

Class — Class uses traits to get functionality.

Reuse — Reduce duplicate code.

Composition — Traits provide composition over inheritance.

1. What is a Trait?

A trait is a mechanism for code reuse in single inheritance languages like PHP. It allows methods to be reused in multiple classes.

```php
<?php
trait MessageTrait {
    public function sayHello() {
        echo "Hello from Trait!";
    }
}
?>
```

Traits cannot be instantiated on their own.

2. Basic Syntax

Use trait inside a class with the use keyword.

```php
<?php
trait MessageTrait {
    public function sayHello() {
        echo "Hello!";
    }
}

class User {
    use MessageTrait;
}
?>
```

use includes all trait methods into the class.

3. Example: Simple Trait

A trait with one method used in a class.

```php
<?php
trait Logger {
    public function log($msg) {
        echo "Log: " . $msg . "\n";
    }
}

class App {
    use Logger;
}

$app = new App();
$app->log("App started");
?>
```

Output: Log: App started

4. Multiple Traits in One Class

A class can use more than one trait.

```php
<?php
trait A {
    public function a() { echo "A"; }
}
trait B {
    public function b() { echo "B"; }
}

class MyClass {
    use A, B;
}

$obj = new MyClass();
$obj->a();   // A
$obj->b();   // B
?>
```

Use comma (,) to include multiple traits.

5. Trait Method Precedence

If same method exists in multiple traits, use insteadof.

```php
<?php
trait First {
    public function say() { echo "First"; }
}
trait Second {
    public function say() { echo "Second"; }
}

class Demo {
    use First, Second {
        Second::say insteadof First;
        First::say as sayFirst;
    }
}
?>
```

Use insteadof to resolve conflicts.

6. Trait with Properties

Traits can also have properties.

```php
<?php
trait Counter {
    public $count = 0;
    public function increment() {
        $this->count++;
    }
}
class Test {
    use Counter;
}

$t = new Test();
$t->increment();
echo $t->count; // 1
?>
```

Key Points

- Yes, traits can have properties.
- Each class has its own copy.

7. Real Life Analogy

Traits are like tools in a toolbox.

- A toolbox (trait) has tools (methods).
- You can take tools and use in any machine (class).
- You can combine multiple toolboxes.

Tools (methods) are shared, machines (classes) remain different.

8. Benefits of Traits

- Promotes code reusability
- Reduces code duplication
- Supports single inheritance with code sharing
- Organizes code better
- Easy to maintain and extend

9. Quick Reference

Term	Description
trait	Defines a reusable set of methods.
use	Includes the trait in a class.
Reusable Methods	Write once, use in many classes.
Multiple Traits	Use more than one trait in a class.
insteadof	Resolve method name conflicts.
as	Alias method to a new name.
Composition	Traits provide composition instead of inheritance.

10. Do's and Don'ts

Do's

- Use traits for code sharing.
- Keep trait methods small and focused.
- Use meaningful names.

Don'ts

- Don't use traits for unrelated methods.
- Don't overload trait with large logic.
- Don't ignore naming conflicts.

11. Trait vs Inheritance

Traits	Inheritance
Horizontal code reuse	Vertical code reuse
Use multiple traits	Single inheritance only
Lightweight	Creates parent-child relationship
Promotes composition	Promotes hierarchy

12. Important Note

Traits are not a replacement for inheritance. They are a way to reuse code in a flexible and clean way using composition.

Pro Tip: Use traits to extract common functionality like logging, validation, or formatting.

Note: PHP traits are available since PHP 5.4.0 and are widely used in modern PHP.

Best Practice: Keep traits simple, focused and reusable. Think: one trait = one responsibility.

Remember: Traits give you the power of reuse without complex inheritance.

PHP NAMESPACES

Organize Code. Avoid Conflicts. Write Better PHP!

```php
<?php
// Namespaces help organize classes and avoid name conflicts
// Use keyword to import classes
// Alias keyword to create shortcuts
>>
```

namespace — Define a named code space.

use — Import classes, functions or constants.

Class Name — Fully qualified names (FQN) identify classes.

Avoid Conflicts — Prevent class name collisions.

Organize Code — Group related code in logical structure.

Import — Bring external classes into current file.

Alias — Create short names for long namespaces.

Structure — Better project structure and maintainability.

1. What is a Namespace?

A namespace is a way to group related classes, interfaces, traits, functions and constants.

```php
<?php
namespace MyApp\Utils;

class Helper {
    public function say() {
        echo "Hello from Helper";
    }
}
>>
```

Namespaces do not affect runtime behavior, they are for code organization.

2. Basic Syntax

Define namespace at the top of the file.

```php
<?php
namespace Vendor\Package\Module;

class Example {
    // class body
}
>>
```

Namespace declaration must be the first statement (except declare).

3. Using Classes with Namespace

Use fully qualified class name or use keyword.

```php
<?php
// Fully Qualified Name
$h = new \MyApp\Utils\Helper();

// Using use keyword
use MyApp\Utils\Helper;
$h = new Helper();
>>
```

Use \\ (backslash) for global namespace.

4. Import Multiple Classes

You can import multiple classes using use.

```php
<?php
use MyApp\Utils\Helper;
use MyApp\Models\User;
use MyApp\Services\MailService;

$helper = new Helper();
$user   = new User();
$mail   = new MailService();
>>
```

Use multiple use statements for different classes.

5. Namespace with Sub-Namespace

Namespaces can be nested (sub-namespaces).

```php
<?php
namespace MyApp\Controllers\Admin;

class DashboardController {
    // controller code
}
>>
```

Sub-namespaces use \\ to separate levels.

6. Alias (Using as)

Create an alias for long namespaces.

```php
<?php
use MyApp\Very\Long\ClassName as Short;

$obj = new Short();
>>
```

Example Aliases

Alias	Equivalent To
as Short	Full class name
as Logger	MyApp\Utils\Logger
as DB	MyApp\Database\Connection

7. Global Namespace

Code not inside any namespace belongs to the global namespace.

```php
<?php
// Global namespace
echo "I am in global namespace";

namespace MyApp;
class Test {}
>>
```

Use \\ to access global namespace from inside a namespace.

8. Real Life Project Structure

Example folder structure using namespaces.

```
src/
  Controllers/
    UserController.php
    AdminController.php
  Models/
    User.php
    Product.php
  Services/
    MailService.php
  Utils/
    Helper.php
```

Example Namespace Mapping

Folder	Namespace
Controllers	MyApp\Controllers
Models	MyApp\Models
Services	MyApp\Services
Utils	MyApp\Utils

9. Best Practices

- Use meaningful namespace names.
- Follow PSR-4 autoloading standard.
- One class per file.
- Use namespaces to avoid naming conflicts.
- Use aliases for long class names.
- Keep namespace structure same as folder structure.

10. Quick Reference

Term	Description
namespace	Define a namespace.
use	Import classes, functions, constants.
Class Name	Fully qualified name (FQN) identifies class.
as	Create alias for a class.
\\	Refers to global namespace.
\ClassName	Access class from global namespace.
Structure	Namespaces organize and structure code.

11. Do's and Don'ts

Do's

- Do use namespaces to organize code.
- Do use use keyword to import classes.
- Do follow PSR-4 autoloading standard.
- Do keep folder and namespace same.

Don'ts

- Don't use same class name in different namespaces without reason.
- Don't forget to import classes you use.
- Don't mix global and namespaced code carelessly.

12. Namespace Cheat Sheet

Task	Syntax
Define Namespace	namespace Vendor\Package;
Use a Class	use Vendor\Package\ClassName;
Use with Alias	use Vendor\Package\ClassName as Alias;
Create Object	$obj = new ClassName();
Global Class	new \ClassName();

13. Important Note

Namespaces are compile-time constructs. They help in organizing large projects and preventing name collisions.

Pro Tip: Use Composer (PSR-4) autoloading to load classes based on namespaces automatically.

Note: Namespaces don't improve performance, they improve code organization.

Best Practice: Always keep your namespace structure clean and follow standard conventions.

Remember: A well-structured namespace today saves you from big conflicts tomorrow!

PHP STATIC & FINAL

Better Performance. Control Behavior. Write Robust Code!

```php
<?php
// static belongs to class, not to objects
// final prevents changes, overrides and inheritance
// Use both smartly for secure and optimized code
?>
```

static — Belongs to class, not to objects.

self:: — Refers to current class scope.

parent:: — Refers to parent class scope.

Static Property — Shared by all objects of the class.

Static Method — Call without object using class name.

final — Prevents change, overrides or inheritance.

Final Class — Cannot be inherited by other classes.

Final Method — Cannot be overridden in child class.

1. What is static?

- static members belong to the class, not to any object.
- Shared among all instances.
- Access using ClassName::member

```php
<?php
class Counter {
    public static $count = 0;
    public static function increment() {
        self::$count++;
    }
}
Counter::increment();
echo Counter::$count;  // 1
?>
```

Static property and method can be accessed without creating an object.

2. Static Property

Shared property across all objects.

```php
<?php
class User {
    public static $total = 0;
    public function __construct() {
        self::$total++;
    }
}
new User();
new User();
echo User::$total;  // 2
?>
```

All objects share the same static property.

3. Static Method

Call static method without object.

```php
<?php
class MathUtil {
    public static function add($a, $b) {
        return $a + $b;
    }
}
echo MathUtil::add(5, 3);  // 8
?>
```

Use ClassName::method() to call static method.

4. self:: vs $this

self:: for static, $this for object.

```php
<?php
class Demo {
    public static $val = 100;
    public $name = "CodeSardar";
    public static function showStatic() {
        echo self::$val;  // OK
    }
    public function showNonStatic() {
        echo $this->name;  // OK
    }
}
Demo::showStatic();  // 100
$obj = new Demo();
$obj->showNonStatic();  // CodeSardar
?>
```

self:: for static context, $this for object context.

5. parent::

Access parent class static members.

```php
<?php
class Base {
    public static function hello() {
        echo "Hello from Base\n";
    }
}
class Child extends Base {
    public static function hi() {
        parent::hello();  // Call parent method
    }
}
Child::hi();  // Hello from Base
?>
```

Use parent:: to access parent's static members.

6. static vs Non-static

Feature	static	Non-static
Belongs to	Class	Object
Access	ClassName::member	$object->member
Memory	One copy for all objects	Each object has its own copy
Use Case	Counters, utilities, config	Object specific data
Keyword	static	No keyword

7. final Keyword

final prevents modification.

Final Class (No Inheritance)

```php
final class Logger {
    // final class
}

class FileLogger extends Logger { }
// Error: Cannot extend final class
```

Final Method (No Override)

```php
class A {
    final public function run() {
        echo "Running...\n";
    }
}
class B extends A {
    public function run() { }
} // Error: Cannot override final method
```

Use final to lock behavior and prevent unwanted changes.

8. Practical Example

Combining static and final.

```php
<?php
final class Config {
    public static $site = "CodeSardar";
    public static function getSite() {
        return self::$site;
    }
    final public function load() {
        echo "Config Loaded\n";
    }
}
echo Config::getSite(); // CodeSardar
$cfg = new Config();
$cfg->load();  // Config Loaded
?>
```

9. Best Practices

- Use static for data shared across all objects.
- Use self:: for static members.
- Use parent:: to access parent class members.
- Use final to prevent inheritance & overriding.
- Don't overuse static, it makes testing hard.
- Keep code clean, maintainable & secure.

10. Quick Reference

Term	Syntax / Usage	Example
Static Property	public static $var	User::$total
Static Method	public static function name()	MathUtil::add(2,3)
self::	self::$var / self::method()	self::$count
$this->	$this->var / $this->method()	$obj->name
parent::	parent::method()	parent::hello()
Final Class	final class ClassName {}	final class Logger {}
Final Method	final function methodName() {}	final function run() {}

11. Do's and Don'ts

Do's

- Do use static for utility methods.
- Do use final for security.
- Do follow good code structure.
- Do use meaningful names.

Don'ts

- Don't access non-static with class name.
- Don't overuse static everywhere.
- Don't break parent behavior.
- Don't ignore code readability.

12. Real Life Analogy

static is like World Settings (shared by all users)

Non-static is like User Settings (different for each user)

13. Important Note

Overusing static makes code tightly coupled and hard to test. Use wisely!

Pro Tip: Use static for helpers, configuration, counters, caches and constant-like data.

Note: PHP constants are also global, but static properties can be changed during runtime.

Best Practice: Prefer dependency injection over static where possible for better testing.

Remember: Final makes your code more secure, stable and predictable.

PHP ABSTRACT CLASSES

Design Blueprint. Force Implementation. Achieve Abstraction!

```php
<?php
// Abstract classes define structure
// Child classes provide implementation
// Helps in clean, maintainable code
?>
```

abstract — Keyword used to define abstraction.

abstract class — A class that cannot be instantiated.

abstract method — A method declared without body.

extends — Child class inherits from abstract class.

Parent Class — Base abstract class (blueprint).

Child Class — Concrete class that provides implementation.

Implementation — Child class must implement all abstract methods.

Abstraction — Hide details, show essential features.

1. What is Abstract Class?

An abstract class is a class that cannot be instantiated on its own. It can have abstract methods (without body) and concrete methods (with body).

```php
<?php
abstract class Animal {
    // Abstract method (no body)
    abstract public function sound();

    // Concrete method
    public function sleep() {
        echo "Animal is sleeping...";
    }
}
?>
```

Abstract class acts as a blueprint for child classes.

2. Abstract Method

Abstract method must be declared in abstract class and must be implemented in child class.

```php
<?php
abstract class Shape {
    abstract public function area(); // No body

    public function info() {
        echo "This is a shape.";
    }
}
?>
```

Child class must provide body for abstract methods.

3. Using Abstract Class

Child class extends abstract class and implements all abstract methods.

```php
<?php
abstract class Shape {
    abstract public function area();
}

class Circle extends Shape {
    public function area() {
        return 3.14 * 5 * 5;
    }
}
?>
```

All abstract methods must be implemented.

4. Cannot Create Object

You cannot create object of abstract class.

This will give Error:

```php
$obj = new Shape();    // Error
```

This is OK:

```php
$obj = new Circle();   // OK
```

Use child class to create objects.

5. Example: Complete

Abstract class with abstract and concrete methods.

```php
<?php
abstract class Vehicle {
    abstract public function move();

    public function fuel() {
        echo "Vehicle needs fuel.";
    }
}

class Car extends Vehicle {
    public function move() {
        echo "Car is moving...";
    }
}

$car = new Car();
$car->move();    // Car is moving...
$car->fuel();    // Vehicle needs fuel.
?>
```

6. Abstract vs Normal Class

Feature	Abstract Class	Normal Class
Instantiation	Cannot create object	Can create object
Methods	Can have abstract methods	All methods must have body
Purpose	Blueprint for other classes	Complete implementation
Keyword	abstract	class
Inheritance	Child must implement abstract methods	Child inherits directly

7. Rules of Abstract Class

- An abstract class must be declared with 'abstract' keyword.
- It can have both abstract and concrete methods.
- It cannot be instantiated.
- Child class must use 'extends' keyword.
- Child class must implement all abstract methods.
- If child class does not implement abstract method, it must also be declared abstract.

8. Multiple Level Inheritance

Abstract class can be extended multiple times.

```php
<?php
abstract class Animal {
    abstract public function sound();
}

abstract class Mammal extends Animal {
    public function breathe() {
        echo "Mammal is breathing.";
    }
}

class Dog extends Mammal {
    public function sound() {
        echo "Dog barks.";
    }
}
?>
```

Animal (Abstract) sound()
↑
Mammal (Abstract) breathe()
↑
Dog (Concrete) sound()

9. Real Life Analogy

Abstract class is like a vehicle blueprint.

- It defines what every vehicle must have.
- Each vehicle (child class) builds it in its own way.
- Same structure, different implementation.

10. Quick Reference

Term	Meaning
abstract	Keyword for abstraction
abstract class	Class cannot be instantiated
abstract method	Method without body
extends	Inheritance keyword
Parent Class	Abstract (base) class
Child Class	Concrete class
Implementation	Provide body for methods
Abstraction	Hide details, show essentials

Pro Tip: Use abstract class when you want to provide a common structure but different implementation.

Note: Abstract classes are useful when you want to enforce a contract for child classes.

Best Practice: Keep abstract classes small and focused. Use meaningful abstract method names.

Important: Child class must implement all abstract methods or it must also be declared abstract.

PHP POLYMORPHISM

One Interface. Many Forms. Dynamic Behavior.

```php
<?php
// Polymorphism = One interface, multiple implementations
// Method calls behave differently based on the object
// Write reusable, flexible and extendable code
>>
```

Polymorphism — Ability to take many forms.

Method Override — Child class redefines parent method.

Parent Type — Reference of parent class or interface.

Child Object — Object of child class assigned to parent type.

Interface — Common contract for multiple classes.

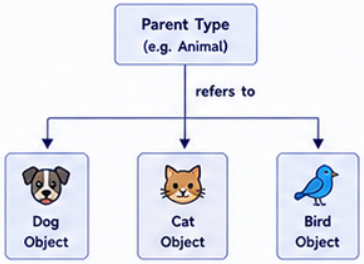
Dynamic Behavior — Behavior decided at runtime.

Reusable Code — Write once, use many times.

OOP — Core concept of Object-Oriented PHP.

1. What is Polymorphism?

Polymorphism allows objects of different classes to be treated as objects of a common parent class or interface.



Same method call, different output.

2. Method Override (Runtime Polymorphism)

Child class overrides parent class method.
The method to be executed is decided at runtime.

```php
<?php
class Animal {
    public function speak() {
        echo "Animal makes sound";
    }
}
class Dog extends Animal {
    public function speak() {
        echo "Dog barks";
    }
}
class Cat extends Animal {
    public function speak() {
        echo "Cat meows";
    }
}
>>
```

Child class provides specific implementation.

3. Polymorphism Example

Parent type reference, child object assigned.

```php
<?php
function animalSound(Animal $a) {
    $a->speak(); // Same call, different output
}

$dog = new Dog();
$cat = new Cat();

$animalSound($dog);  // Dog barks
$animalSound($cat);  // Cat meows
>>
```

Output:

```
Dog barks
Cat meows
```

Same function call `$a->speak()`, behavior changes based on object.

4. Using Interface (Polymorphism)

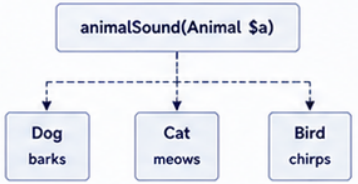
Interfaces allow multiple classes to provide their own implementation.

```php
<?php
interface Shape {
    public function area();
}
class Circle implements Shape {
    public function area() {
        return 3.14 * 5 * 5;
    }
}
class Rectangle implements Shape {
    public function area() {
        return 5 * 10;
    }
}
function printArea(Shape $s) {
    echo "Area: " . $s->area() . "\n";
}
printArea(new Circle());    // Area: 78.5
printArea(new Rectangle()); // Area: 50
>>
```

Different classes, same interface, same function call.

5. Benefits of Polymorphism

- Increases code reusability
- Makes code more flexible and extendable
- Reduces duplicate code
- Easy to add new classes
- Improves maintainability
- Follows OOP principles



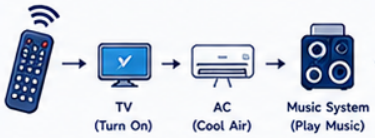
6. Types of Polymorphism

Compile Time (Static Polymorphism)	Run Time (Dynamic Polymorphism)
• Method Overloading	• Method Overriding
• Operator Overloading	• Interface Implementation
• Resolved at compile time	• Resolved at runtime

PHP supports Run Time Polymorphism.

7. Real Life Analogy

A remote control (parent type) can control different devices (child objects). The button (method call) is same, but action (behavior) is different.



8. Polymorphism in OOP

- Polymorphism is a key feature of OOP.
- It promotes abstraction and loose coupling.
- Write programs for interfaces, not implementations.
- Focus on what an object does, not how.

9. Best Practices

- Use interfaces for common behavior.
- Always prefer polymorphism over if-else or switch.
- Keep classes small and focused.
- Code for flexibility, not for one case.

10. Quick Reference

Term	Meaning
Polymorphism	Ability to take many forms.
Method Override	Child class provides new implementation of parent method.
Parent Type	Reference of parent class/interface.
Child Object	Object of child class.
Interface	Contract that multiple classes follow.
Dynamic Behavior	Behavior decided at runtime.
Reusable Code	Write once, use in many forms.

Pro Tip: Program to an interface, not an implementation.

Note: Polymorphism works only with inheritance and interfaces.

Best Practice: Always override methods with same signature.

Code Smarter: Use polymorphism to make your code clean, flexible and future-ready.

Remember: Focus on behavior, not on object type.

PHP MAGIC METHODS

Special Methods. Automatic Trigger. Powerful Features!

```php
<?php
// Magic methods are called automatically by PHP
// when certain events occur.
// They start and end with double underscores.
>>
```

__construct() — Called when object is created.

__destruct() — Called when object is destroyed.

__get() — Called when trying to read inaccessible property.

__set() — Called when trying to write inaccessible property.

__isset() — Called when isset() or empty() is used.

__unset() — Called when unset() is used on property.

__call() — Called when calling inaccessible method.

__toString() — Called when object is treated as a string.

1. __construct()

Automatically called when object is created.

```php
<?php
class User {
    public function __construct($name) {
        echo "Hello, $name! Object created.";
    }
}

$user = new User("CodeSardar");

>>
```

Example Output:

```
Hello, CodeSardar! Object created.
```

Use Case: Initialization, DB connection, setup.

2. __destruct()

Automatically called when object is destroyed.

```php
<?php
class User {
    public function __destruct() {
        echo "Object is destroyed.";
    }
};
$user = new User();

>>
```

Example Output:

```
Object is destroyed.
```

Use Case: Cleanup, close files, release memory.

3. __get()

Called when reading inaccessible property.

```php
<?php
class User {
    private $name = "CodeSardar";
    public function __get($prop) {
        echo "Reading '$prop' is not allowed!";
    }
}
$user = new User();
echo $user->name;
>>
```

Example Output:

```
Reading 'name' is not allowed!
```

Use Case: Virtual properties, lazy loading.

4. __set()

Called when writing to inaccessible property.

```php
<?php
class User {
    private $name;
    public function __set($prop, $value) {
        echo "Setting '$prop' to '$value'
            is not allowed!";
    }
}
$user = new User();
$user->name = "John";
>>
```

Example Output:

```
Setting 'name' to 'John' is not allowed!
```

Use Case: Validation, access control.

5. __isset()

Called when isset() or empty() is used.

```php
<?php
class User {
    private $name;
    public function __isset($prop) {
        echo "Checking if '$prop' is set!";
        return isset($this->$prop);
    }
}
$user = new User();
isset($user->name);
>>
```

Example Output:

```
Checking if 'name' is set!
```

Use Case: Check property existence.

6. __unset()

Called when unset() is used on property.

```php
<?php
class User {
    private $name = "CodeSardar";
    public function __isset($prop) {
        echo "Unsetting '$prop' is not allowed!";
    }
}
$user = new User();
unset($user->name);
>>
```

Example Output:

```
Unsetting 'name' is not allowed!
```

Use Case: Prevent removing important data.

7. __call()

Called when calling inaccessible method.

```php
<?php
class User {
    public function __call($method, $args) {
        echo "Method '$method' does not exist!";
    }
}
$user = new User();
$user->sayHello();
>>
```

Example Output:

```
Method 'sayHello' does not exist!
```

Use Case: Handle undefined methods, APIs.

8. __toString()

Called when object is treated as a string.

```php
<?php
class User {
    private $name = "CodeSardar";
    public function __toString() {
        return "User: {$this->name}";
    }
}
$user = new User();
echo $user;  // object as string
>>
```

Example Output:

```
User: CodeSardar
```

Use Case: String representation, logging.

9. Practical Example (All Together)

```php
<?php
class Product {
    private $name;
    public function __construct($name) { $this->name = $name; }
    public function __destruct() { echo "Product '{$this->name}' destroyed."; }
    public function __get($prop) {  echo "Cannot read '$prop'!"; }
    public function __set($prop, $value) {  "Cannot set '$prop'!"; }
    public function __isset($prop) { echo "Is '$prop' set?s->$prop); }
    public function __unset($prop) {  echo "Cannot unset '$prop'!"; }
    public function __call($method, $args) { echo "Method '$method' not found"; }
    public function __toString() {  return "Product: {$this->name}"; }
}
$p = new Product("Laptop"); echo $p; >>
```

Output:

```
Product: Laptop

Product 'Laptop' destroyed.
```

10. Do's and Don'ts

Do's

- Use magic methods only when necessary.
- Keep logic simple and clear.
- Use __construct() for initialization.
- Use __toString() carefully.
- Document your magic methods.

Don'ts

- Don't overuse magic methods.
- Don't break expected behavior.
- Don't use for performance critical code.
- Don't return wrong data types.
- Don't ignore access control.

11. Quick Reference Table

Method	Trigger	Purpose
__construct()	Object creation	Initialization
__destruct()	Object destruction	Cleanup
__get()	Read inaccessible property	Get virtual property
__set()	Write inaccessible property	Set & validate property
__isset()	isset() or empty()	Check property
__unset()	unset()	Prevent unset
__call()	Call inaccessible method	Handle undefined methods
__toString()	Object as string	String representation

12. Why Magic Methods?

- **Automatic** — No need to call manually.
- **Flexible** — Handle dynamic behavior.
- **Reusable** — Write once, use many times.
- **Powerful** — Core of advanced PHP (OOP).

Pro Tip: Magic methods are powerful but use them wisely. Don't make code confusing.

Note: All magic methods are optional. Use only when needed.

Best Practice: Keep code clean, readable and maintainable. Follow OOP principles.

Remember: Magic methods start with double underscore __ (e.g., __get()).

PHP TYPE DECLARATIONS

Strong Code. Clear Intent. Fewer Bugs!

```php
<?php
// Type declarations define the kind of data
// a variable, parameter, or return value can hold.
// They help catch errors early and improve code quality.
>>
```

int — Whole numbers. (e.g., 10, -5)

float — Decimal numbers. (e.g., 3.14, -0.5)

string — Sequence of characters. (e.g., "Hello")

bool — Boolean values. true or false

array — Ordered collection of values.

object — Instance of a class. (User, DateTime)

mixed — Can be any type of value.

void — No value is returned.

return type — Specifies the type a function returns.

1. int (Integer)

Stores whole numbers without decimals.

```php
<?php
function add(int $a, int $b): int {
    return $a + $b;
}

$result = add(5, 10);
echo $sreult;   // 15
```

Example Output

```
15
```

Range: ~-9E18 to 9E18 (platform dependent)

2. float (Floating Point)

Stores decimal numbers.

```php
<?php
function area(float $r): float {
    return 3.1416 * $r * $r;
}

echo area(2.5);    // 19.635
```

Example Output

```
19.635
```

Precision: depends on platform (usually 64-bit)

3. string (String)

Stores sequence of characters.

```php
<?php
function greet(string $name): string {
    return "Hello, " . $name . "!";
}

echo greet("CodeSardar");
```

Example Output

```
Hello, CodeSardar!
```

Use single or double quotes.

4. bool (Boolean)

Stores true or false values.

```php
<?php
function isAdult(int $age): bool {
    return $age >= 18;
}

var_dump(isAdult(20));   // true
var_dump(isAdult(15));   // false
```

Example Output

```
bool(true)
bool(false)
```

Case-insensitive: true, TRUE, True

5. array (Array)

Stores multiple values in ordered list.

```php
<?php
function getNames(): array {
    return ["Ram", "Shyam", "Mohan"];
}

print_r(getNames());
```

Example Output

```
Array
(
    [0] => Ram
    [1] => Shyam
    [2] => Mohan
)
```

Can hold different data types too.

6. object (Object)

Stores instance of a class.

```php
<?php
class User {
    public string $name;
    public function __construct($name) {
        $this->name = $name;
    }
}
function getUser(): object {
    return new User("Jeet");
}
var_dump(getUser());
```

Example Output

```
object(User)#1 (1) {
  ["name"]=>
  string(4) "Jeet"
}
```

Must be an instance of a class.

7. mixed (Mixed)

Can accept any type of value.

```php
<?php
function printValue(mixed $value): void {
    var_dump($value);
}
printValue(10);
printValue("Hello");
printValue([1, 2, 3]);
```

Example Output

```
int(10)
string(5) "Hello"
array(3) { [0]=> int(1) ... }
```

Use when type is unknown.

8. void (Void)

Function does not return any value.

```php
<?php
function sayHi(string $name): void {
    echo "Hi, $name!\n";
}

sayHi("Jeet");
```

Example Output

```
Hi, Jeet!
```

Use : void in return type.

9. Return Type Examples

Functions can declare the type of value they return.

Function	Return Type	Example Call	Output
function sum(int $a, int $b): int	15	sum(5,10)	15
function pi(): float	3.1416	pi()	3.1416
function welcome(): string	Welcome!	welcome()	Welcome!
function isLogin(): bool	true	isLogin()	true
function getList(): array	[1,2,3]	getList()	Array(3)
function getUser(): object	User Object	getUser()	object(User)
function anything(): mixed	Any value	anything()	...
function doWork(): void	No output	doWork()	(nothing)

10. Nullable Types (?Type)

Allow null in addition to the specified type.

```php
<?php
function getUser(?string $name): ?string {
    if ($name === null) return null;
    return "Hello, $name";
}

var_dump(getUser("Jeet"));
var_dump(getUser(null));
```

Example Output

```
string(12) "Hello, Jeet"
NULL
```

Use ? before type to allow null.

11. Union Types (PHP 8+)

Allow multiple types.

```php
<?php
function printId(int|string $id): void {
    echo "ID: $id";
}

printId(101);    // int
printId("A101"); // string
```

Example Output

```
ID: 101
ID: A101
```

Use | to separate types.

12. Type Declaration Rules

- Types are checked at runtime.
- Mismatched types cause TypeError.
- Strict types can be enabled.
- Applies to parameters, return types and properties (PHP 7.4+).
- Makes code readable and maintainable.
- Helps in IDE autocompletion.

13. Quick Reference Table

Type	Example	Description
int	10	Whole numbers
float	3.14	Decimal numbers
string	"Hello"	Text values
bool	true / false	Boolean values
array	[1, 2, 3]	Ordered collection
object	new User()	Class instance
mixed	10 / "Hi" / []	Any type
void	(no value)	No return value

Pro Tip: Always use type declarations. They save time and prevent bugs!

Note: Enable strict types using: `declare(strict_types=1);`

Best Practice: Use the most specific type possible. Avoid mixed unless necessary.

Code Smarter: Type declarations + Good logic = Powerful PHP!

Remember: Types make your code safe, clear and future-ready!

PHP ANONYMOUS FUNCTIONS

Small Function. Big Power. Write Less. Do More!

```php
<?php
// Anonymous function (closure)
// Stored in a variable and used as needed
// Powerful for callbacks and scope control
>>
```

Anonymous Function — Function without a name.

Closure — Works with variable scope.

function() — Created using function() keyword.

use — Import variables from outer scope.

Parameters — Accept values like normal functions.

Return — Return values using return statement.

Callback — Used in functions that accept another function.

Variable Scope — Access outer variables using use.

1. What is Anonymous Function?

An anonymous function is a function without a name. It can be stored in a variable and used later.

```php
<?php
$greet = function() {
    echo "Hello, CodeSardar!";
};

$greet();  // Hello, CodeSardar!
>>
```

Useful for short, one-time logic.

2. With Parameters

Anonymous functions can accept parameters like normal functions.

```php
<?php
$add = function($a, $b) {
    return $a + $b;
};

echo $add(5, 7);  // 12
>>
```

Parameters behave same as normal functions.

3. Using use (Variable Scope)

Use 'use' to access variables from outer scope inside the function.

```php
<?php
$name = "CodeSardar";
$show = function() use ($name) {
    echo "Hello, $name!";
};

$show();  // Hello, CodeSardar!
>>
```

Without use, outer variables are not accessible.

4. Return Values

Anonymous functions can return values using return statement.

```php
<?php
$multiply = function($a, $b) {
    return $a * $b;
};

$result = $multiply(4, 6);
echo $rseult;  // 24
>>
```

Return makes it reusable in expressines.

5. As Callback Function

Anonymous functions are commonly used as callbacks.

```php
<?php
$numbers = [1, 2, 3, 4, 5];

$squares = array_map(function($n) {
    return $n * $n;
}, $numbers);

print_r($squares);
// [1, 4, 9, 16, 25]
>>
```

Great for array_map, usort, etc.

6. Stored in Variable

Anonymous functions can be stored in variables and passed around.

```php
<?php
$sayHello = function($name) {
    return "Hi, $name";
};

echo $sayHello("Jeet");
// Hi, Jeet
>>
```

Functions are first-class citizens in PHP.

7. Multiple use Variables

You can import multiple variables into the closure.

```php
<?php
$x = 10; $y = 20;
$sum = function() use ($x, $y) {
    return $x + $y;
};

echo $sum();  // 30
>>
```

Separate multiple variables with comma.

8. Without use (Scope Error)

If you try to access outer variable without use, it will not work.

```php
<?php
$name = "CodeSardar";
$show = function() {
    // echo $name; // Error
    echo "Name is not accessible!";
};

$show();
>>
```

PHP will throw undefined variable error.

9. Practical Examples

A. Sort Array with usort()

```php
$users = [
  ['name' => 'Ram', 'age' => 25],
  ['name' => 'Shyam', 'age' => 20],
  ['name' => 'Mohan', 'age' => 30]
];

usort($users, function($a, $b) {
   return $a['age'] <=> $b['age'];
});

print_r($users);
```

Sort by age using anonymous function.

B. Filter Array

```php
$numbers = [1, 2, 3, 4, 5, 6];

$even = array_filter($numbers, function($n) {
    return $n % 2 == 0;
});

print_r($even);
// [2, 4, 6]
```

Filter even numbers.

C. Delay Execution (Concept)

```php
$task = function($msg) {
    echo "Task: $msg\n";
};

// Later...

$task("Backup Completed");
// Task: Backup Completed
```

Store now, run later.

10. Quick Reference Table

Term	Meaning
Anonymous Function	Function without a name.
Closure	Anonymous function with scope.
function()	Keyword used to create closure.
use	Access outer variables.
Parameters	Accept input values.
Return	Return output value.
Callback	Passed to another function.
Variable Scope	Controlled using use keyword.

Pro Tip: Use anonymous functions for short, one-time tasks and callbacks.

Note: Closures do not exist in global scope by default.

Best Practice: Keep anonymous functions small and focused.

Code Smarter: Less code, more power with closures!

Remember: Use 'use' to avoid scope and undefined variable errors.

PHP CALLBACKS

Pass a Function. Execute Later. Powerful & Flexible!

```php
<?php
// Callbacks allow you to pass a function as an argument
// and call it later when needed.
// Built-in functions that accept callbacks make
// your code clean, reusable and powerful.
```

Callback — A function passed as an argument.

call_user_func() — Calls a user defined function dynamically.

array_map() — Applies callback to each array element.

array_filter() — Filters array using callback condition.

array_reduce() — Reduces array to a single value.

Function — Named function used as callback.

Closure — Anonymous function used as callback.

Callable — Any valid callback: function, closure, array.

1. call_user_func()

Calls the given function with the provided arguments.

```php
<?php
function sayHello($name) {
    return "Hello, $name!";
}

$result = call_user_func('sayHello', 'Jeet');
echo $result;   // Hello, Jeet!
>>
```

Output

```
Hello, Jeet!
```

Useful when function name is stored in a variable.

2. array_map()

Applies callback to each element and returns new array.

```php
<?php
$numbers = [1, 2, 3, 4, 5];
$double = function($n) {
    return $n * 2;
};

$result = array_map($double, $numbers);
print_r($result);
// [2, 4, 6, 8, 10]
>>
```

Output

```
[2, 4, 6, 8, 10]
```

Transforms each element using callback.

3. array_filter()

Filters array elements using callback condition.

```php
<?php
$numbers = [1, 2, 3, 4, 5, 6, 7];
$even = function($n) {
    return $n % 2 == 0;
};

$result = array_filter($numbers, $even);
print_r($result);
// [2, 4, 6]
>>
```

Output

```
[2, 4, 6]
```

Keeps elements that return true.

4. array_reduce()

Reduces array to a single value using callback.

```php
<?php
$numbers = [1, 2, 3, 4];
$sum = function($carry, $item) {
    return $carry + $item;
};

$result = array_reduce($numbers, $sum, 0);
echo $result;   // 10
>>
```

Output

```
10
```

Accumulates result into single value.

5. Common Callbacks

You can pass function name, closure or array.

```php
<?php
function square($n) { return $n * $n; }
$fn = 'square';
$result1 = call_user_func($fn, 5);   // 25

$result2 = call_user_func((function($n){
    return $n * $n;
}, 6);  // 36

$obj = new Math();
$result3 = call_user_func([$obj, 'cube'], 3);
// 27
>>
```

Output

```
25, 36, 27
```

Any callable is acceptable.

6. Built-in Functions that Use Callbacks

Function	Use
call_user_func()	Calls a user function dynamically.
call_user_func_array()	Calls a function with parameters as array.
array_map()	Applies callback to each element.
array_filter()	Filters array using callback.
array_reduce()	Reduces array to single value.
usort() / uasort()	Sorts array using callback.
array_walk()	Applies callback to each element (by ref).

Many PHP functions support callbacks!

7. Practical Example

Square all even numbers and get total sum

```php
<?php
$numbers = [1, 2, 3, 4, 5, 6, 7, 8];

$even = array_filter($numbers, function($n) {
    return $n % 2 == 0;
});
$squared = array_map(function($n) {
    return $n * $n;
}, $even);
$total = array_reduce($squared, function($carry, $item) {
    return $carry + $item;
}, 0);

echo "Total: $total";   // Total: 120
>>
```

8. Callback Flow

Data / Array `[1, 2, 3, 4]` → Callback `Function / Closure` → Result `[Processed Data]`

Flow:

1. Pass function as argument.
2. PHP stores and calls it when needed.
3. Callback processes the data.
4. Result is returned.

9. Callable Types

- **Function Name** — `'functionName'`
- **Closure** — `function($x) { ... }`
- **Array (Class Method)** — `['ClassName', 'method']`
- **Object Method** — `[$object, 'method']`

All are valid callbacks!

10. Best Practices

- Use callbacks to write reusable code.
- Prefer closures for short, one-time logic.
- Keep callbacks small and focused.
- Validate input when using call_user_func().
- Use built-in functions with callbacks for performance.

Pro Tip: Callbacks make your code modular, reusable and easy to extend.

Note: A callback is any callable that can be called like a function.

Best Practice: Use array_map(), array_filter(), array_reduce() for clean code.

Code Smarter: Less loop, more callback! Write smart, not hard.

Remember: Callbacks run your code when and how you want.

PHP CLOSURES

Small Function. Big Power. Remember the Context!

```php
<?php
// A closure is an anonymous function
// that can use variables from parent scope.
// It's powerful, flexible and widely used.
// Pass it, return it, and use it anywhere.
>>
```

function() — Creates a closure using function().

use — Import variables from outer scope.

Variable — Access parent variables safely.

Scope — Remembers the scope where it was created.

Callback — Can be passed as callback to functions.

Anonymous Function — Closure is an anonymous function.

Return — Closures can be returned from functions.

1. What is Closure?

A closure is an anonymous function that has access to variables from its parent scope.

```php
<?php
$msg = "Hello, CodeSardar!";
$hello = function() use ($msg) {
    echo $msg;
};
$hello();  // Hello, CodeSardar!
>>
```

Closure "remembers" variables from where it was created.

2. Using use to Access Variables

Use the use keyword to access outer variables inside closure.

```php
<?php
$name = "Jeet";
$greet = function() use ($name) {
    echo "Hi, $name!";
};
$greet();  // Hi, Jeet!
>>
```

Without use, variable is not accessible inside closure.

3. Modifying Variables in Closure

Use & to modify the outer variable inside the closure.

```php
<?php
$count = 0;
$inc = function() use (&$count) {
    $count++;
};
$inc(); $inc();
echo $count;  // 2
>>
```

Pass by reference (&) to modify outer variable.

4. Closure as Callback

Closures are widely used as callbacks in functions.

```php
<?php
$numbers = [1, 2, 3, 4];
$squares = array_map(function($n) {
    return $n * $n;
}, $numbers);
print_r($squares);
// [1, 4, 9, 16]
>>
```

Closure passed to array_map() as callback.

5. Returning a Closure

A function can return a closure.

```php
<?php
function makeMultiplier($factor) {
    return function($num) use ($factor) {
        return $num * $factor;
    };
}
$double = makeMultiplier(2);
echo $double(5);  // 10
>>
```

Function returns a closure that can be used later.

6. Multiple Variables

You can import multiple variables using use.

```php
<?php
$x = 10; $y = 20;
$sum = function() use ($x, $y) {
    return $x + $y;
};
echo $sum();  // 30
>>
```

Import many variables: use ($x, $y, $z)

7. Closure & Callback Example

Using closure with array_filter() as callback.

```php
<?php
$nums = [1, 2, 3, 4, 5, 6];
$even = array_filter($nums, function($n) {
    return $n % 2 == 0;
});
print_r($even);
// [2, 4, 6]
>>
```

Closure used as callback to filter even numbers.

8. Closure vs Normal Function

Feature	Normal Function	Closure
Name	Has a name	No name (anonymous)
Scope Access	No access to parent scope	Access parent scope using use
Can be Assigned	No	Yes (to variable)
Can be Returned	No	Yes
Common Use	General purpose	Callbacks, factories, handlers

9. Practical Example

```php
<?php
function operation($a, $b, $callback) {
    return $callback($a, $b);
}

$add = function($x, $y) { return $x + $y; };
$mul = function($x, $y) { return $x * $y; };

echo operation(5, 3, $add);  // 8
echo operation(5, 3, $mul);  // 15
>>
```

Output

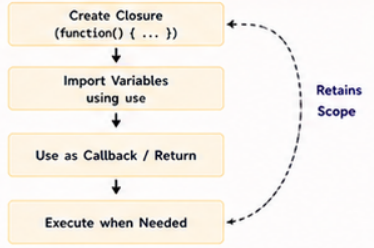
```
8
15
```

Closures make code flexible and reusable!

10. Common Built-in Functions Using Closures

Function	Use of Closure (Callback)
array_map()	Applies function to each array element
array_filter()	Filters array using a condition
array_reduce()	Reduces array to a single value
usort() / uasort()	Sorts array using custom logic
preg_replace_callback()	Uses closure to process matches
call_user_func()	Calls a closure or function dynamically

11. Closure Life Cycle



Pro Tip: Closures are perfect for short, reusable, and dynamic code.

Note: Closures inherit scope, but only variables passed with use.

Best Practice: Keep closure small and focused. Use meaningful variable names.

Code Smarter: Closures + Callbacks = Powerful PHP Development!

Remember: Don't forget use or you'll get unexpected results!

PHP GENERATORS

Lazy Load. Save Memory. Generate Values on Demand!

```php
<?php
// Generators make iteration simple & efficient
// using yield and yield from.
// They remember state between calls.
>>
```

Generator — Special function that can pause execution.

yield — Returns a value and pauses.

yield from — Delegate values from another iterable.

Iterator — Generators are Iterator objects.

Memory Efficient — Values generated only when needed.

Lazy Execution — Generate values only on demand.

Values — Produce sequence of values easily.

Iteration — Easy to loop with foreach.

1. Basic Generator

A generator function uses yield instead of return.

```php
<?php
function countUp($limit) {
    for ($i = 1; $i <= $limit; $i++) {
        yield $i;
    }
}
foreach (countUp(5) as $num) {
    echo $num . " ";
}
>>
```

Output

```
1 2 3 4 5
```

2. Using yield

yield returns value and pauses execution.

```php
<?php
function letters() {
    yield 'A';
    yield 'B';
    yield 'C';
}
foreach (letters() as $l) {
    echo $l . " ";
}
>>
```

Output

```
A B C
```

3. yield with Key & Value

You can yield key => value pairs.

```php
<?php
function users() {
    yield 1 => "Ram";
    yield 2 => "Shyam";
    yield 3 => "Mohan";
}
foreach (users() as $id => $name) {
    echo "$id => $name\n";
}
>>
```

Output

```
1 => Ram
2 => Shyam
3 => Mohan
```

4. yield from (Delegation)

Delegate to another generator.

```php
<?php
function numbers() {
    yield 1; yield 2; yield 3;
}
function moreNumbers() {
    yield from numbers();
    yield 4; yield 5;
}
foreach (moreNumbers() as $n) {
    echo $n . " ";
}>
```

Output

```
1 2 3 4 5
```

5. Iterator Behavior

Generators are iterable objects (Iterator).

```php
<?php
$gen = (function() {
    yield 'X';
    yield 'Y';
    yield 'Z';
})();
foreach ($gen as $v) {
    echo "$v ";
}
>>
```

Output

```
X Y Z
```

6. Large Data (Memory Efficient)

Generate big data without high memory use.

```php
<?php
function bigData($n) {
    for ($i = 1; $i <= $n; $i++) {
        yield $i; // only when needed
} }

foreach (bigData(1000000) as $v) {
    if ($v > 5) break; // stop early
    echo $v . " ";
}
>>
```

Output

```
1 2 3 4 5
```

7. Generator vs Normal Function

Feature	Normal Function	Generator
Return Type	Single value	Multiple values (one by one)
Memory Use	High (if large array)	Low (on demand)
Execution	Runs completely	Pauses & resumes
Keyword	return	yield
Use	Small data	Large/Infinite data

8. Practical Example (Fibonacci)

```php
<?php
function fibonacci($n) {
    $a = 0; $b = 1;
    for ($i = 0; $i < $n; $i++) {
        yield $a;
        [$a, $b] = [$b, $a + $b];
    }
}
foreach (fibonacci(7) as $f) {
    echo $f . " ";
}
>>
```

Output

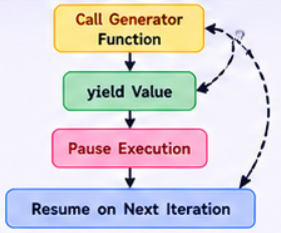
```
0 1 1
2 3 5 8
```

Memory Efficient!

9. When to Use Generators

- When working with large datasets
- When reading files line by line
- When generating infinite sequences
- When you need lazy evaluation
- When performance is important

10. Generator Flow



11. Quick Reference

- function name() { yield value; }
- yield key => value;
- yield from iterable;
- foreach (generator() as $val) {}
- Generators return Traversable (Iterator).

Pro Tip: Use generators for huge data processing to save memory!

Note: Generators automatically implement Iterator interface.

Best Practice: Prefer yield and yield from for clean, efficient code.

Code Smarter: Lazy loading = Fast & Smart PHP Development!

Remember: Generators make your code powerful and memory friendly!

PHP ITERATORS

Iterate Smartly. Code Cleanly. Loop Powerfully!

```php
<?php
// Iterators provide a way to access elements
// of a collection one at a time.
// They follow a standard interface.
// Perfect for custom collections!
>>
```

Iterator — Defines methods to traverse elements.

current() — Returns current element.

key() — Returns the key of the current element.

next() — Moves pointer to the next element.

rewind() — Rewinds pointer to the first element.

valid() — Checks if current position is valid.

foreach — Easiest way to iterate values.

Iterable — Any object that can be iterated.

Collection — A group of items stored together.

1. What is an Iterator?

An Iterator provides a standard way to traverse a collection of elements.

```php
<?php
$numbers = new ArrayIterator([10, 20, 30, 40]);

foreach ($numbers as $num) {
    echo $num . " ";
}
// Output: 10 20 30 40
>>
```

Iterators hide the internal structure of the collection.

2. current()

Returns the current element in the collection.

```php
<?php
$it = new ArrayIterator(['a','b','c']);

echo $it->current();  // a

$it->next();
echo $it->current();  // b
>>
```

Does not move pointer.

3. key()

Returns the key of the current element.

```php
<?php
$it = new ArrayIterator(['x'=>'X','y'=>'Y','z'=>'Z']);

echo $it->key();  // x

$it->next();
echo $it->key();  // y
>>
```

Keys can be integers or strings.

4. next()

Moves the pointer to the next element.

```php
<?php
$it = new ArrayIterator([1,2,3]);
echo $it->current();  // 1
$it->next();
echo $it->current();  // 2
$it->next();
echo $it->current();  // 3
>>
```

Use next() to advance forward.

5. rewind()

Rewinds the pointer to the first element.

```php
<?php
$it = new ArrayIterator([100,200,300]);
$it->next();
$it->next();
echo $it->current();  // 300
$it->rewind();
echo $it->current();  // 100
>>
```

Brings pointer back to the start.

6. valid()

Checks if the current position is valid.

```php
<?php
$it = new ArrayIterator(['a','b']);
$it->next();  // a
$it->next();  // b
$it->next();  // beyond end
var_dump($it->valid()); // false
>>
```

Returns true if current position is valid.

7. foreach Loop

The easiest way to iterate over any iterable object.

```php
<?php
$it = new ArrayIterator(['p','q','r']);

foreach ($it as $key => $value) {
    echo "$key => $value\n";
}
>>
```

Works with all Iterable objects.

8. Iterable

Anything that can be looped over is called Iterable.

```php
<?php
function printItems(iterable $items) {
    foreach ($items as $item) {
        echo $item . " ";
    }
}

printItems([5,10,15]);  // 5 10 15
>>
```

Arrays, objects can be Iterable.

9. Custom Iterator Example (Simple Class)

You can create your own iterator by implementing Iterator interface.

```php
<?php
class MyCollection implements Iterator {
    private $data = [10, 20, 30];
    private $position = 0;
    public function current(): mixed { return $this->data[$this->position]; }
    public function key(): mixed { return $this->position; }
    public function next(): void { $this->position++; }
    public function rewind(): void { $this->position = 0; }
    public function valid(): bool { return isset($this->data[$this->position]); }
}
$coll = new MyCollection();
foreach ($coll as $key => $value) {
    echo "$key => $value\n";
}
>>
```

Output

```
0 => 10
1 => 20
2 => 30
```

Custom iterators give you full control over iteration.

Iterator vs Iterable vs foreach

Feature	Iterator	Iterable	foreach
Definition	Interface with methods for iteration	Any object that can be looped over	Language construct for easy iteration
Purpose	Define how to iterate	Mark object as iterable	Iterate over iterables
Example	ArrayIterator, Generator	Array, Traversable object	foreach ($arr as $v) {}
Requires Methods	Yes (6 methods)	No (just Traversable)	No

Quick Method Summary

Method	Description
current()	Returns current element
key()	Returns key of current element
next()	Moves pointer to next element
rewind()	Rewinds pointer to first element
valid()	Checks if current position is valid

Common Iterator Classes in PHP

Class	Description
ArrayIterator	Iterates over arrays
ArrayObject	Array + object features
IteratorIterator	Wraps another iterator
RecursiveIteratorIterator	Iterates recursively
Generator	Uses yield for iteration

Pro Tip: Use iterators for large data sets or when you need custom iteration logic.

Note: All iterator objects must implement Iterator or IteratorAggregate.

Best Practice: Use foreach for simple cases and custom iterators for advanced needs.

Code Smarter: Iterators make your code modular, reusable and efficient.

Remember: Always check valid() before accessing current() or key().

PHP PDO CHEAT SHEET

Powerful. Secure. Flexible. Database Access Made Easy!

```php
<?php
// PDO (PHP Data Objects) provides a
// consistent way to access databases.
// Supports prepared statements,
// prevents SQL Injection & more!
>>
```

Database Connection — Create a secure connection to any supported database.

DSN — Data Source Name defines the connection details.

prepare() — Prepares an SQL statement for execution.

execute() — Executes a prepared statement.

query() — Executes a query directly.

fetch() — Fetches the next row from the result set.

fetchAll() — Fetches all rows from the result set.

1. Database Connection

Create a connection to the database using PDO.

```php
<?php
$dsn = "mysql:host=localhost;
 dbname=mydb;charset=utf8mb4";
$user = "root";
$pass = "password";
try {
   $pdo = new PDO($dsn, $user, $pass);
   $pdo->setAttribute(PDO::ATTR_ERRMODE,
      PDO::ERRMODE_EXCEPTION);
   echo "Connected successfully!";
} catch (PDOException $e) {
   echo "Connection failed: " . $e->getMessage();
}
>>
```

Use try...catch to handle connection errors.

2. DSN (Data Source Name)

DSN format depends on the database driver.

```php
<?php
// MySQL
$dsn = "mysql:host=localhost;
dbname=mydb;charset=utf8mb4";

// SQLite
$dsn = "sqlite:database.db";

// PostgreSQL
$dsn = "pgsql:host=localhost;
 dbname=mydb";
>>
```

Always specify charset for better encoding support.

3. prepare()

Prepares an SQL statement for execution.

```php
<?php
$stmt = $pdo->prepare(
  "SELECT * FROM users WHERE
  id = :id"
);

?>
```

Use placeholders (:) to prevent SQL Injection.

4. execute()

Executes a prepared statement with optional parameters.

```php
<?php
$stmt = $pdo->prepare(
  "SELECT * FROM users WHERE
  id = :id"
);
$stmt->execute(['id' => 1]);
>>
```

Returns true on success, false on failure.

5. query()

Executes a query directly and returns a statement.

```php
<?php

$stmt = $pdo->query("SELECT *
FROM users");

>>
```

Use for simple queries without parameters.

6. fetch()

Fetches the next row from the result set.

```php
<?php
$stmt = $pdo->query("SELECT *
FROM users");

$row = $stmt->fetch(PDO::FETCH_ASSOC);

if ($row) {
   echo $row['name'];
}
>>
```

FETCH_ASSOC returns an associative array.

7. fetchAll()

Fetches all rows from the result set.

```php
<?php
$stmt = $pdo->query("SELECT *
FROM users");

$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);

foreach ($rows as $row) {
   echo $row['name'] . "<br>";
}
>>
```

Returns an array of all rows. Great for multiple results.

PDO Fetch Modes

Mode	Description
PDO::FETCH_ASSOC	Returns an associative array
PDO::FETCH_NUM	Returns a numeric array
PDO::FETCH_BOTH	Returns both associative and numeric arrays
PDO::FETCH_OBJ	Returns an anonymous object
PDO::FETCH_CLASS	Returns an object of specified class

Complete Workflow Example

```php
<?php
$pdo = new PDO("mysql:host=localhost;dbname=mydb;charset=utf8mb4",
  "root", "password");
$stmt = $pdo->prepare("SELECT id, name, email FROM users WHERE status = :status");
$stmt->execute(['status' => 'active']);
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);
foreach ($users as $user) {
   echo $user['id'] . " - " . $user['name'] . " (" . $user['email'] . ")<br>";
}
>>
```

Output

1 - Alice (alice@example.com)
2 - Bob (bob@example.com)
3 - Charlie (charlie@example.com)

Prepared statements keep your code safe and clean!

Common PDO Exceptions

Exception	Reason
PDOException	Connection or query error
SQLSTATE[HY000]	General error
SQLSTATE[42S22]	Column not found
SQLSTATE[23000]	Integrity constraint violation
SQLSTATE[42000]	Syntax error or access violation

Best Practices

- Always use prepared statements.
- Use exception handling.
- Validate and sanitize input.
- Close connections when not needed.
- Use fetch modes as per requirement.
- Keep sensitive data (passwords) secure.

Pro Tip: Use prepared statements every time. They are safer and more efficient.

Note: PDO supports MySQL, SQLite, PostgreSQL, SQL Server, Oracle and more.

Best Practice: Set PDO::ATTR_ERRMODE to PDO::ERRMODE_EXCEPTION for better debugging.

Code Smarter: Write secure, clean and maintainable database code with PDO.

Remember: Never trust user input. Always validate before using in queries.

PHP MySQL CONNECTION

Connect PHP with MySQL using PDO – Secure, Simple & Powerful!

```php
<?php
// PHP + MySQL (PDO) Connection
// Secure. Fast. Reliable.
>>
```

Host
Server address where MySQL is running.

Database
Name of the database you want to connect.

Username
MySQL username with access permission.

Password
Password for the MySQL username.

PDO
PHP Data Objects for secure and flexible connection.

Connection
Establishes a connection with the database.

Connection Details (Example)

Host	localhost
Database	test_db
Username	root
Password	your_password
PDO	php_pdo_mysql
Connection	Successful

Quick Test Query

```php
<?php
$stmt = $pdo->query("SELECT * FROM users LIMIT 5");
$rows = $stmt->fetchAll();

foreach ($rows as $row) {
    echo $row['id'] . " - " . $row['name'] . "<br>";
}
>>
```

Output (Example)

```
1 - John
2 - Alice
3 - Bob
```

PDO Connection Code

```php
<?php
// MySQL Connection using PDO

$host   = 'localhost';
$db     = 'test_db';
$user   = 'root';
$pass   = 'your_password';
$charset = 'utf8mb4';

$dsn = "mysql:host=$host;dbname=$db;charset=$charset";
$options = [
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    PDO::ATTR_EMULATE_PREPARES => false,
];

try {
    $pdo = new PDO($dsn, $user, $pass, $options);
    echo "✅ Connection successful!";
} catch (PDOException $e) {
    echo "❌ Connection failed: " . $e->getMessage();
}
>>
```

DSN Format

```
mysql:host=HOST;
dbname=DATABASE;
charset=CHARSET
```

Options Used

```
ERRMODE_EXCEPTION
FETCH_ASSOC
EMULATE_PREPARES
```

How It Works

1. **Set Connection Details** – Define host, database name, username and password.
2. **Create DSN** – Build the DSN string with host, database and charset.
3. **Set Options** – Set PDO options for error handling and fetch mode.
4. **Create PDO Object** – Create a new PDO instance with DSN, username, password and options.
5. **Handle Exceptions** – Use try....catch to handle connection errors gracefully.
6. **Ready to Use** – Connection established! Now you can run queries and manage data.

Pro Tip Always use PDO with prepared statements to protect against SQL Injection.

Note Use strong passwords and limited database users for better security.

Best Practice Keep connection code in a separate file and include it where needed.

Code Smarter PDO is database independent. You can switch databases by changing only the DSN.

Remember Never expose database credentials in public files or version control.

PHP PREPARED STATEMENTS

Write Once, Execute Many Times – Secure, Fast & Efficient!

```php
<?php
// Prepared Statements in PDO
// Protects from SQL Injection
// Improves Performance
// Reuse, Bind & Execute!
>>
```

prepare() — Prepares an SQL statement for execution.

execute() — Executes the prepared statement with bound data.

bindParam() — Binds a variable by reference to a named or positional placeholder.

bindValue() — Binds a value to a placeholder at the time of execution.

Placeholders — Named (:name) or Positional (?) placeholders.

Parameters — Values passed to the placeholders (user input safe).

SQL Injection — Prevents malicious SQL by separating code and data.

Secure Query — Safe, reliable and recommended way to run queries.

1. prepare()

Prepares an SQL statement with placeholders.

```php
<?php
$sql = "SELECT * FROM users
WHERE email = :email AND
status = ?";
$stmt = $pdo->prepare($sql);
?>
```

Prepares once, can execute many times.

2. bindParam()

Binds a variable by reference.

```php
<?php
$email = "john@example.com";
$status = 1;
$stmt->bindParam(':email', $email,
  PDO::PARAM_STR);
$stmt->bindParam(1, $status,
  PDO::PARAM_INT);
>>
```

Changes in variable will be reflected on execute().

3. bindValue()

Binds a value directly.

```php
<?php
$id = 10;
$stmt = $pdo->prepare(
  "DELETE FROM users WHERE
  id = :id");
$stmt->bindValue(':id', $id,
  PDO::PARAM_INT);
>>
```

Best when value is known at the time of binding.

4. execute()

Executes the prepared statement.

```php
<?php
$success = $stmt->execute();
if ($success) {
  echo "Query executed
   successfully!";
} else {
  echo "Execution failed!";
}
>>
```

Returns true on success, false on failure.

5. Placeholders

Use named (:name) or positional (?) placeholders.

Named Placeholder

```
SELECT * FROM users
WHERE name = :name
AND age > :age;
```

Positional Placeholder

```
INSERT INTO users (name, age)
VALUES (?, ?);
```

Named is more readable, positional is shorter.

6. Parameters

Pass safe values as parameters.

```php
<?php
$name = "Alice";
$age  = 25;
$stmt->execute([
  ':name' => $name,
  ':age'  => $age
]);
>>
```

Keeps data separate from SQL query.

7. SQL Injection

Prevents SQL Injection.

```php
<?php
// UNSAFE (Vulnerable)
$id = $_GET['id'];
$pdo->query("SELECT * FROM
users WHERE id = $id");

// SAFE (Prepared Statement)
$stmt = $pdo->prepare("SELECT *
FROM users WHERE id = ?");
$stmt->execute([$id]);
>>
```

Never mix user input directly in SQL query.

8. Secure Query

Prepared statements are secure & efficient.

- Precompiled SQL
- Secure from SQL Injection
- Reusable
- Better Performance
- Clean & Maintainable Code

Complete Example: Select Data

```php
<?php
$pdo = new PDO("mysql:host=localhost;dbname=testdb;charset=utf8", "root", "password");
$sql = "SELECT id, name, email FROM users WHERE status = :status ORDER BY name";
$stmt = $pdo->prepare($sql);
$status = 1;
$stmt->bindParam(':status', $status, PDO::PARAM_INT);
$stmt->execute();
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);
foreach ($users as $user) {
   echo $user['id'] . " - " . $user['name'] . " (" . $user['email'] . ")<br>";
}
>>
```

Output (Example)

1 - John (john@example.com)

2 - Alice (alice@example.com)

3 - Bob (bob@example.com)

Common Methods

Method	Description
prepare($sql)	Prepares an SQL statement
execute([$params])	Executes the prepared statement
bindParam($param, $var, $type)	Binds a variable by reference
bindValue($param, $value, $type)	Binds a value directly
fetch()	Fetches next row (default: PDO::FETCH_BOTH)
fetchAll($mode)	Fetches all rows
rowCount()	Returns number of affected rows
errorInfo()	Returns error information

Placeholder Types

Type	Syntax	Example
Named	:name	:email, :id, :status
Positional	?	?, ?, ?
Mixed	:name and ?	Allowed, but not recommended

PDO Data Types

Type	Constant
Integer	PDO::PARAM_INT
String	PDO::PARAM_STR
Boolean	PDO::PARAM_BOOL
Null	PDO::PARAM_NULL
LOB	PDO::PARAM_LOB

Why Use Prepared Statements?

Security — Prevents SQL injection attacks

Performance — Precompiled & faster execution

Reusability — Execute multiple times easily

Clean Code — More readable & maintainable

Pro Tip: Always use prepared statements when dealing with user input.

Note: Do not use prepare() for queries without user input.

Best Practice: Bind parameters with proper data types.

Code Smarter: Use named placeholders for complex queries.

Remember: Security First, Always! Keep your queries safe.

CRUD OPERATIONS

Create, Read, Update, Delete – Manage Data Easily!

- Simple Queries
- Secure with PDO
- Best Practices
- Clean & Efficient Code

C — CREATE
Add new data to the database.

R — READ
Retrieve data from the database.

U — UPDATE
Modify existing data in the database.

D — DELETE
Remove data from the database.

users

id	name	email	age
1	John	john@example.com	25
2	Alice	alice@example.com	22
3	Bob	bob@example.com	28

CREATE (INSERT)

Add new records into the table.

SQL

```
INSERT INTO users (name, email, age)
VALUES (:name, :email, :age);
```

PHP (PDO)

```
$stmt = $pdo->prepare(
  "INSERT INTO users (name, email, age)
   VALUES (:name, :email, :age)");
$stmt->bindValue(':name', $name);
$stmt->bindValue(':email', $email);
$stmt->bindValue(':age', $age);
$stmt->execute();
// New record inserted successfully!
```

Use INSERT to add new data.

READ (SELECT)

Retrieve data from the table.

SQL

```
SELECT * FROM users WHERE id = :id;
```

PHP (PDO)

```
$stmt = $pdo->prepare(
  "SELECT * FROM users WHERE id = :id" );

$stmt->bindValue(':id', $id, PDO::PARAM_INT);
$stmt->execute();
$row = $stmt->fetch(PDO::FETCH_ASSOC);

// Data fetched successfully!
```

Use SELECT to read or fetch data.

UPDATE

Update existing records.

SQL

```
UPDATE users SET name = :name,
        email = :email, age = :age
    WHERE id = :id;
```

PHP (PDO)

```
$stmt = $pdo->prepare(
 "UPDATE users SET name = :name,
        email = :email, age = :age
        WHERE id = :id'");
$stmt->bindValue(':name', $name);
$stmt->bindValue(':email', $email);
$stmt->bindValue(':age', $age, PDO::PARAM_INT);
$stmt->bindValue(':id', $id, PDO::PARAM_INT);
$stmt->execute();
// Record updated successfully!
```

Use UPDATE to modify existing data.

DELETE

Delete records from the table.

SQL

```
DELETE FROM users WHERE id = :id;
```

PHP (PDO)

```
$stmt = $pdo->prepare(
  "DELETE FROM users WHERE id = :id" );

$stmt->bindValue(':id', $id, PDO::PARAM_INT);
$stmt->execute();

// Record deleted successfully!
```

Use DELETE to remove data.

Secure
Always use prepared statements to prevent SQL Injection.

Efficient
PDO is fast, flexible and database independent.

Best Practice
Validate inputs and handle errors properly.

Keep It Clean
Write clean, readable and maintainable code.

Remember
Test your queries and always backup your database.

SQL QUERIES in PHP

Write Powerful Queries. Manage Data Easily!

```php
<?php
// PHP + MySQL = Perfect Combo!
// Fast, Secure & Efficient
// Use PDO for Better Security
>>
```

SELECT — Retrieve data from database.

INSERT — Insert new data into database.

UPDATE — Update existing data in database.

DELETE — Delete data from database.

WHERE — Filter records using condition.

JOIN — Combine rows from multiple tables.

ORDER BY — Sort results in ascending/descending.

LIMIT — Limit number of results.

SELECT Query

Get data from table.

```sql
SELECT * FROM users;
```

```php
<?php
$stmt = $pdo->query("SELECT * FROM users");
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);

foreach ($users as $user) {
  echo $user['name'] . " - " . $user['email'];
}
>>
```

Use SELECT to read data.

INSERT Query

Add new record to table.

```sql
INSERT INTO users (name, email, age)
VALUES ('John', 'john@example.com', 25);
```

```php
<?php
$stmt = $pdo->prepare(
    "INSERT INTO users (name, email, age)
     VALUES (:name, :email, :age)",
);
$stmt->execute([
    ':name' => 'John',
    ':email' => 'john@example.com',
    ':age' => 25
]);
>>
```

Use INSERT to add new data.

UPDATE Query

Modify existing records.

```sql
UPDATE users SET age = 26 WHERE id = 1;
```

```php
<?php
$stmt = $pdo->prepare(
   "UPDATE users SET age = :age WHERE id = :id",
);
$stmt->execute([
   ':age' => 26,
   ':id'  => 1
]);
>>
```

Use UPDATE to modify data.

DELETE Query

Remove records from table.

```sql
DELETE FROM users WHERE id = 1;
```

```php
<?php
$stmt = $pdo->prepare(
  "DELETE FROM users WHERE id = :id"
);
$stmt->execute([
  ':id' => 1
]);
>>
```

Use DELETE to remove data.

WHERE Clause

Filter your results.

```sql
SELECT * FROM users WHERE age > 18
AND status = 'active';
```

```php
<?php
$stmt = $pdo->query(
    "SELECT * FROM users WHERE age > 18
     AND status = 'active'",
);
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
>>
```

JOIN Example

Combine data from multiple tables.

```sql
SELECT u.name, p.title FROM users u
INNER JOIN posts p ON u.id = p.user_id;
```

```php
<?php
$stmt = $pdo->query(
   "SELECT u.name, p.title
    FROM users u
    INNER JOIN posts p ON u.id = p.user_id"
);
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
>>
```

ORDER BY

Sort your results.

```sql
SELECT * FROM users ORDER BY name ASC;
```

```php
<?php
$stmt = $pdo->query(
  "SELECT * FROM users ORDER BY name ASC",
);
>>
```

ASC = A to Z, DESC = Z to A

LIMIT

Limit number of results.

```sql
SELECT * FROM users LIMIT 5;
```

```php
<?php
$stmt = $pdo->preare(
  "SELECT * FROM users LIMIT 5",
);
>>
```

Useful for pagination.

</> Combined Example

All together in one query.

```sql
SELECT * FROM users
WHERE age > 18 ORDER BY name DESC LIMIT 10;
```

```php
<?php
$stmt = $pdo->query(
   "SELECT * FROM users
    WHERE age > 18
    ORDER BY name DESC LIMIT 10"
);
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);
>>
```

Sample Table: users

id	name	email	age	status
1	John	john@example.com	25	active
2	Alice	alice@example.com	22	active
3	Bob	bob@example.com	28	inactive
4	Mike	mike@example.com	30	active
5	Sara	sara@example.com	19	active

Common WHERE Conditions

=	Equal	age = 25
>	Greater Than	age > 18
<	Less Than	age < 30
>=	Greater or Equal	age >= 18
<=	Less or Equal	age <= 30
LIKE	Search Pattern	name LIKE 'J%'
IN	In a List	id IN (1,2,3)

Pro Tip

- Always use Prepared Statements.
- Never trust user input.
- Validate and sanitize data.

Best Practice

- Use PDO for secure connection.
- Keep queries clean & optimized.
- Handle errors properly.

Example Connection (PDO)

```php
$pdo = new PDO("mysql:host=localhost;dbname=test_db;charset=utf8",
"root", "password",[
[PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
```

Note

PDO is faster, secure and database independent.

Remember

Good Queries = Good Performance!

DATABASE TRANSACTIONS

Ensure Data Integrity & Reliability with PHP PDO Transactions!

```php
<?php
// Transactions in PHP (PDO)
// Safe. Reliable. Consistent.
?>
```

TRANSACTION
A sequence of one or more SQL queries treated as a single unit.

beginTransaction()
Starts a new transaction.

commit()
Saves all changes permanently.

rollBack()
Undoes changes if something goes wrong.

RECOVERY
Allows system to recover from failures.

DATABASE
Transactions keep your database safe & stable.

ACID PRINCIPLES

A — ATOMICITY
All operations in a transaction succeed or none at all.

C — CONSISTENCY
Transaction brings database from one valid state to another.

I — ISOLATION
Transactions are independent and do not interfere with each other.

D — DURABILITY
Once committed, changes are permanent even after failure.

WHEN TO USE TRANSACTIONS?

- Money Transfer (Debit/Credit)
- Order Processing
- Multi-step operations
- When data must remain consistent

TRANSACTION WORKFLOW

Start Transaction
beginTransaction()
→ Execute Queries
(Multiple SQL Queries)
→ If Success
commit()
→ If Failed
rollBack()

EXAMPLE: MONEY TRANSFER

```php
<?php
try {
    $pdo->beginTransaction();
    $pdo->prepare("UPDATE accounts SET balance = balance - ? WHERE id = ?")
        ->execute([$amount, $from_id]);

    $pdo->prepare("UPDATE accounts SET balance = balance + ? WHERE id = ?")
        ->execute([$amount, $to_id]);

    $pdo->commit();
    echo "Transaction committed succesfully!";
} catch (Exception $e) {
    $pdo->rollBack();
    echo "Transaction failed! Changes rolled back.";
}
```

Explanation

If both UPDATE queries succeed, changes are committed.

If any query fails, all changes are rolled back.

TRANSACTION FUNCTIONS

beginTransaction()	Starts a new transaction.
commit()	Commits the current transaction.
rollBack()	Rolls back the current transaction.

EXAMPLE SCENARIO

User A — Balance: ₹1000 → Transfer ₹200 → User B — Balance: ₹500

Success	A: ₹800	B: ₹700	commit()
Failure	A: ₹1000	B: ₹500	rollBack()

PRO TIP Keep transactions short. Long transactions can lock resources.

NOTE Always handle exceptions to avoid partial updates.

BEST PRACTICE Use transactions only when necessary for better performance.

CODE SMARTER Transactions protect data integrity and build trust.

REMEMBER Backup your database regularly for recovery!

PASSWORD HASHING

Secure Passwords with PHP

- Secure Password Storage
- Protect Users & Data
- Best Practices
- Strong Authentication

password_hash() — Creates a secure hash of the password.

password_verify() — Verifies a password against the hash.

PASSWORD_DEFAULT — Uses the default algorithm (bcrypt recommended).

HASH — The encrypted output stored in database.

VERIFY — Checks if the given password matches the hash.

PASSWORD — The original plain text password (never store it).

SECURITY — Hashing protects passwords from being stolen.

AUTHENTICATION — Used during login to verify user credentials.

1. Create Password Hash

Use `password_hash()` to hash a password before storing in the database.

```php
<?php
$password = "MySecureP@ss123";
$hash = password_hash($password, PASSWORD_DEFAULT);
echo $hash;
?>
```

Output (Hash Example)

```
$2y$10$X9uN6kzY8l6R2sTzZp6hH.u8QwQwKz3J6mQpZ8xYkV1FzQxY8l6R2sT
```

About PASSWORD_DEFAULT

- Uses the best algorithm supported by PHP.
- Currently uses bcrypt.
- Automatically upgrades in future PHP versions.

2. Verify Password

Use `password_verify()` to check the password during login.

```php
<?php
$input_password = "MySecureP@ss123";
$hash = '$2y$10$X9uN6kzY8l6R2sTzZp6hH
.u8QwQwKz3J6mQpZ8xYkV1FzQxY8l6R2sT';
if (password_verify($input_password, $hash)) {
   echo "Login Successful!";
} else {
   echo "Invalid Password!";
}
?>
```

Output

- Login Successful!
- Invalid Password!

3. Complete Example (Register & Login)

Register (Hash & Store)

```php
<?php
$password = $_POST['password'];
$hash = password_hash($password, PASSWORD_DEFAULT);
// Store $hash in database
?>
```

Login (Verify)

```php
<?php
$password = $_POST['password'];
$stored_hash = $row['password_hash']; // from DB
if (password_verify($password, $stored_hash)) {
   echo "Login Successful!";
  // Start session
} else {
   echo "Invalid Password!";
}
?>
```

Password Hashing Summary

Feature	Description
password_hash()	Hashes the password securely.
password_verify()	Verifies the password with hash.
PASSWORD_DEFAULT	Uses the default secure algorithm.
Hash	Encrypted password (store in DB).
Password	Plain text (never store).
Security	Protects against data breaches.
Authentication	Confirms user identity during login.

Best Practices

- Never store plain text passwords.
- Always use password_hash().
- Always use password_verify() for login.
- Use PASSWORD_DEFAULT for future proof.
- Use HTTPS for data transfer.

How It Works

User creates account → password_hash() creates hash → Hash stored in database → During login password_verify() checks hash → Access Granted / Access Denied

Why Hash Passwords?

- Protects passwords if database is stolen.
- Prevents reverse engineering.
- Adds extra layer of security.

Pro Tip

Bcrypt (PASSWORD_DEFAULT) automatically adds salt and handles everything for you!

Important

Never try to decode or decrypt hashes. Hashing is one-way and cannot be reversed.

Quick Example

```
$hash = password_hash("MyPassword123", PASSWORD_DEFAULT);  // Store this in DB
if (password_verify("MyPassword123", $hash)) { echo "OK"; }  // Verify during login
```

Secure Today, Safe Tomorrow!

AUTHENTICATION

Secure Login & User Management with PHP

Login | Register | Session | Cookie | Password Hash | User | Logout

AUTHENTICATION FLOW

User → Register → Login → Authentication Success → Session Created → Logout (Session End)

Authentication verifies the user's identity before giving access to protected resources.

WHAT IS AUTHENTICATION?

Authentication is the process of verifying who a user is.

- Confirms user identity
- Protects data & resources
- Uses sessions, cookies & secure passwords

KEY COMPONENTS

Login	User enters credentials to access account.
Register	New user creates an account.
Session	Stores user data on server during login.
Cookie	Stores small data on user's browser.
Password Hash	Password stored in encrypted (hashed) form.
User	Registered user in the system.
Logout	Ends user session and logs out.

USERS TABLE (Example)

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT
  CURRENT_TIMESTAMP
);
```

1. REGISTER (Create Account)

Save user with hashed password.

```php
<?php
$name  = $_POST['name'];
$email = $_POST['email'];
$pass = password_hash($_POST['password'],
                      PASSWORD_DEFAULT);
$stmt = $pdo->prepare("INSERT INTO users
      (name,email,password)
      VALUES (,?,?)");
$stmt->execute([$name,$email,$pass]);
echo "Registration Successful!";
?>
```

Tip: Always hash passwords before storing in database.

2. LOGIN (User Login)

Verify user & start session.

```php
<?php
$email = $_POST['email'];
$pass  = $_POST['password'];
$stmt  = $pdo->prepare("SELECT * FROM users
         WHERE email = ?");
$stmt  execute([$email]);
$user  = $stmt->fetch(PDO::PETCH_ASSOC);
if ($user && password_verify($pass, $user['password'])) {
    $_SESSION['user_id'] = $user['id'];
    $_SESSION['name']    = $user['name'];
    echo "Login Successful!";
    header("Location: dashboard.php");
  else {
    echo "Invalid Email or Password!";
}?>
```

Note: Use password_verify() to check password.

3. SESSION (Server-Side)

Store user data during login.

```php
<?php
session_start();

// After successful login
$_SESSION['user_id']  = $user['id'];
$_SESSION['name']     = $user['name'];
?>
```

Benefits:

- Data stored on server
- More secure than cookies
- Ends when user logs out

4. COOKIE (Client-Side)

Store small data on user browser.

```php
<?php
// Set cookie (expires in 7 days)
setcookie("user_email", $user['email'],
  time() + (7*24*60*60), "/", "", true, true);
?>
```

Benefits:

- Stored on client browser
- Can be persistent
- Use for "Remember Me" feature

5. LOGOUT (End Session)

Destroy session and cookies.

```php
<?php
session_start();
session_unset();
session_destroy();
setcookie("user_email", "", time()-3600, "/");
header("Location: login.php");
exit();
?>
```

Tip: Always logout user on public computers.

PASSWORD HASHING (Security First)

- Never store plain passwords.
- Always use password_hash().
- Use PASSWORD_DEFAULT (Bcrypt algorithm).

Create Hash

```php
$pass  = "MySecurePass@123";
$hash  = password_hash($pass, PASSWORD_DEFAULT);
// Store $hash in database
```

→

Verify Password

```php
$pass  = $_POST['password'];
$hash  = $user['password'];
if (password_verify($pass, $hash)) {
    echo "Password Matched!";
}
```

BEST PRACTICES

- Always use HTTPS.
- Hash passwords using PASSWORD_DEFAULT.
- Use prepared statements to prevent SQL Injection.
- Regenerate session ID after login.
- Limit login attempts (Brute Force Protection).
- Logout and destroy session properly.

AUTHENTICATION AT A GLANCE

User → Authenticate (Login) → Session Created → Access Granted → Logout (Session End)

Strong Authentication = Secure Application

Pro Tip — Security is not optional, it's essential!

Secure Passwords | Safe Sessions | Protected Data | Happy Users

FILE UPLOAD

Upload Files Easily & Securely in PHP

UPLOAD | $_FILES | VALIDATION | SECURE | MOVE FILE

1. UPLOAD FORM (HTML)

```
<form action="upload.php" method="post"
 enctype="multipart/form-data">
   <label for="file">Choose File:</label>
   <input type="file" name="file" id="file" required>
   <br><br>
   <button type="submit" name="upload">Upload</button>
</form>
```

2. $_FILES SUPERGLOBAL

When a file is uploaded, PHP stores file information in the $_FILES superglobal array.

Key	Description
name	Original name of the uploaded file.
type	MIME type of the uploaded file.
size	Size of the uploaded file in bytes.
tmp_name	Temporary name of the file on server.
error	Error code associated with the upload. (0 = No error)

3. MOVE UPLOADED FILE

Use move_uploaded_file() to move the uploaded file from temporary folder to desired location.

```php
<?php
$target_dir = "uploads/";
$target_file = $target_dir . basename($_FILES["file"]["name"]);
if (move_uploaded_file($_FILES["file"]["tmp_name"], $target_file)) {
    echo "File uploaded successfully!";
} else {
    echo "Failed to move uploaded file.";
}
?>
```

4. $_FILES EXAMPLE OUTPUT

```
Array
(
    [file] => Array
        (
            [name] => photo.jpg
            [type] => image/jpeg
            [size] => 102400
            [tmp_name] =>
             C:\xampp\tmp\phpA1B2.tmp
            [error] => 0
        )
)
```

UPLOAD PROCESS FLOW

5. VALIDATION (Important)

- Check file type (MIME)
- Check file size
- Check for upload errors
- Allow only specific extensions
- Avoid dangerous files

Common Errors (error code)

0 = No error, file uploaded successfully.
1 = Uploaded file exceeds upload_max_filesize.
2 = Uploaded file exceeds MAX_FILE_SIZE.
3 = File was only partially uploaded.
4 = No file was uploaded.
6 = Missing a temporary folder.
7 = Failed to write file to disk.
8 = PHP extension stopped the upload.

6. VALIDATION EXAMPLE

```php
<?php
$allowed = array('image/jpeg', 'image/png', 'image/gif');
$max_size = 2 * 1024 * 1024; // 2MB

$file = $_FILES['file'];
$file_type = $file['type'];
$file_size = $file['size'];

if (!in_array($file_type, $allowed)) {
    die('Invalid file type. Only JPG, PNG, GIF allowed.');
}
if ($file_size > $max_size) {
    die('File size must be less than 2MB.');
}
if ($file['error'] !== UPLOAD_ERR_OK) {
    die('File upload error. Code: ' . $file['error']);
}
?>
```

7. COMPLETE UPLOAD EXAMPLE (upload.php)

```php
<?php
if (isset($_POST['upload'])) {
    $allowed = ['image/jpeg', 'image/png', 'image/gif'];
    $max_size = 2 * 1024 * 1024; // 2MB
    $file = $_FILES['file'];
    if (in_array($file['type'], $allowed) && $file['size'] <= $max_size && $file['error'] == 0) {
        $target_dir = "uploads/";
        if (!file_exists($target_dir)) { mkdir($target_dir, 0755, true); }
        $target_file = $target_dir . time() . '_' . basename($file['name']);

        if (move_uploaded_file($file['tmp_name'], $target_file)) {
            echo "<p style='color:green;'>File uploaded successfully!</p>";
            echo "File: " . htmlspecialchars($file['name']);
        } else {
            echo "<p style='color:red;'>Failed to move uploaded file.</p>";
        }
    } else {
        echo "<p style='color:red;'>Invalid file or upload error.</p>";
    }
?>
```

8. RECOMMENDED EXTENSIONS

9. IMPORTANT INI SETTINGS

Setting	Recommended Value
file_uploads	On
upload_max_filesize	8M
post_max_size	8M
max_file_uploads	20

10. BEST PRACTICES

- Always validate file type and size.
- Rename files before storing to avoid conflicts.
- Store files outside the public directory if possible.
- Scan files for malware (if needed).
- Limit upload size in php.ini.
- Use HTTPS for secure uploads.

PRO TIP Always validate on both client-side (JS) and server-side (PHP).

NOTE Never trust user uploads. Validate and sanitize always!

SECURITY Do not allow PHP or executable files. Restrict upload types.

BONUS Use unique file names (time() or UUID) to prevent overwriting.

REST API

Build Scalable APIs with PHP

Endpoint API URL | HTTP Protocol | JSON Data Format | Secure & Scalable | Fast Performance

HTTP METHODS

GET	Retrieve data from the server.
POST	Send data to the server to create a resource.
PUT	Update an existing resource (replace completely).
PATCH	Update an existing resource (partial update).
DELETE	Delete a resource from the server.

API ENDPOINT EXAMPLE

METHOD	ENDPOINT	DESCRIPTION
GET	/api/users	Get all users
POST	/api/users	Create a new user
PUT	/api/users/{id}	Update user (full)
PATCH	/api/users/{id}	Update user (partial)
DELETE	/api/users/{id}	Delete a user

Base URL Example: https://api.example.com

REQUEST & RESPONSE EXAMPLE

Request (POST /api/users)

```
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN

{
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "age": 32
}
```

Response (201 Created)

```
HTTP/1.1 201 Created
Content-Type: application/json
{
  "success": true,
  "message": "User created successfully",
  "data": {
    "id": 101,
    "name": "Lovejeet Singh",
    "email": "lovejeet@example.com",
    "age": 32,
    "created_at": "2025-08-14T10:30:00Z"
  }
}
```

JSON DATA FORMAT

JSON Request Example

```
{
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "age": 32,
  "skills": ["PHP", "MySQL", "API"],
  "active": true
}
```

JSON

JSON Response Example

```
{
  "success": true,
  "data": {
    "id": 101,
    "name": "Lovejeet Singh",
    "email": "lovejeet@example.com"
  },
  "message": "Request successful"
}
```

JSON is lightweight and easy to read, perfect for APIs.

HTTP STATUS CODES

CODE	STATUS	DESCRIPTION
200	OK	Request successful.
201	Created	Resource created successfully.
400	Bad Request	Invalid request or parameters.
401	Unauthorized	Authentication required.
403	Forbidden	Access denied.
404	Not Found	Resource not found.
500	Internal Server Error	Something went wrong on server.

PHP REST API EXAMPLE (GET /api/users)

```
<?php
header("Content-Type: application/json");
$users = [
    ["id" => 1, "name" => "Lovejeet Singh", "email" => "lovejeet@example.com"],
    ["id" => 2, "name" => "Aman Kumar", "email" => "aman@example.com"]
];
echo json_encode([
    "success" => true,
    "data" => $users,
    "message" => "Users fetched successfully"
]);
?>
```

BEST PRACTICES

- Use meaningful endpoints.
- Use appropriate HTTP methods.
- Always return proper status codes.
- Validate input data.
- Use HTTPS for security.
- Return consistent JSON format.

API FLOW

Client → Request (HTTP Method) → API Server (Process) → Database → JSON Response

QUICK TIPS

- GET → Read
- POST → Create
- PUT/PATCH → Update
- DELETE → Remove

Build Fast. Scale Easy. Go RESTful!

PHP PROJECT STRUCTURE

Organized • Scalable • Maintainable

Organized Code | Easy to Maintain | Scalable Application | Secure & Clean | Better Performance

PROJECT FOLDER STRUCTURE

```
my-php-project/
├── public/          Publicly accessible files (Document Root)
├── src/             Application source code
│   ├── controllers/ Handle HTTP requests
│   ├── models/      Database models & business logic
│   ├── views/       UI templates (PHP/HTML)
│   └── routes/      Define application routes
├── config/          Configuration files
├── uploads/         Uploaded files storage
├── .env             Environment variables
├── composer.json    Composer dependencies
├── .htaccess        Apache configuration
└── README.md        Project documentation
```

FOLDER EXPLANATION

public/	Entry point of the application. Contains index.php, assets (CSS, JS, images).
src/	All core application code.
config/	Application configuration files. (DB connection, App settings, etc.)
controllers/	Handle user requests and return responses.
models/	Interact with database and contain business logic.
views/	UI templates and pages.
routes/	Define and manage application routes.
uploads/	Store user uploaded files (images, docs, etc.).

ENTRY POINT (public/index.php)

```php
<?php
declare(strict_types=1);

require_once __DIR__ . '/../vendor/autoload.php';
require_once __DIR__ . '/../config/app.php';

$app = require_once __DIR__ . '/../src/bootstrap.php';
$app->run();
?>
```

WHY THIS STRUCTURE?

- Separation of concerns
- Easy to maintain and update
- Scalable for large applications
- Secure (Core files are not publicly accessible)
- PSR-4 Autoloading ready

REQUEST FLOW

User Request (Browser)
↓
public/index.php (Entry Point)
↓
routes/ (Define Route)
↓
controllers/ (Handle Request)
↓
models/ (Business Logic)
↓
views/ (Render Response)
↓
Response to User

EXAMPLE FILES

routes/web.php

```php
<?php
$router->get('/', [
  'controller' => 'HomeController',
  'action'     => 'index'
]);

$router->get('/about', [
  'controller' => 'HomeController',
  'action'     => 'about'
]);
```

controllers/HomeController.php

```php
<?php
class HomeController {
    public function index() {
        $title = "Welcome";
        return view('home/index',
            ['title' => $title]);
    }

    public function about() {
        return view('home/about');
    }
}
```

models/User.php

```php
<?php
class User {
    public function all() {
        // Fetch all users
        return $this->db
            ->query("SELECT *
              FROM users")
            ->fetchAll();
    }
}
```

views/home/index.php

```php
<!DOCTYPE html>
<html>
<head>
    <title><?= $title ?></title>
</head>
<body>
    <h1>Welcome to My App</h1>
    <p>This is home page.</p>
</body>
</html>
```

config/database.php

```php
<?php
return [
    'host'    => 'localhost',
    'db'      => 'my_database',
    'user'    => 'root',
    'pass'    => 'password',
    'charset' => 'utf8mb4'
];
```

PUBLIC DIRECTORY (Accessible to Web)

- Only public/ directory is accessible from the web.
- Keep sensitive files outside public/ for security.

Document Root
- public/
  - index.php
  - assets/
  - .htaccess

IMPORTANT FILES

.env	Environment variables
composer.json	PHP dependencies
.htaccess	URL rewriting & security
README.md	Project information

BEST PRACTICES

- Use .env for environment variables
- Never expose config files in public
- Follow PSR-4 autoloading standard
- Keep code clean and modular
- Validate and sanitize all inputs

TIP

This structure works great with Composer, PSR-4 Autoloading, and modern PHP frameworks!

TECHNOLOGIES

PHP 8+ | Composer | MySQL